**JSX:**

Это JSX — расширение языка JavaScript.

JSX производит «элементы» React.

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

По умолчанию React DOM [экранирует](https://stackoverflow.com/questions/7381974/which-characters-need-to-be-escaped-on-html) все значения, включённые в JSX перед тем как отрендерить их. Это гарантирует, что вы никогда не внедрите чего-либо, что не было явно написано в вашем приложении.

Babel компилирует JSX в вызовы React.createElement().

Примеры эквивалентны:

Изображение выглядит как текст

Автоматически созданное описание

**Рендеринг элементов:**

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

Изображение выглядит как текст

Автоматически созданное описание

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" \t "_blank)(неизменны). После создания элемента нельзя изменить его потомков или атрибуты.

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 JavaScript. Они принимают произвольные входные данные (так называемые «пропсы») и возвращают React-элементы, описывающие, что мы хотим увидеть на экране.

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

**Изображение выглядит как текст

Автоматически созданное описание**

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.

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.

Например, компонент App может отрендерить компонент Welcome несколько: **Изображение выглядит как текст

Автоматически созданное описание**

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).

Изображение выглядит как текст

Автоматически созданное описание

Такие функции называют [«чистыми»](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D1%82%D0%BE%D1%82%D0%B0_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8), потому что они не меняют свои входные данные и предсказуемо возвращают один и тот же результат для одинаковых аргументов.

**React-компоненты обязаны вести себя как чистые функции по отношению к своим пропсам.**

**Состояние и жизненный цикл:**

Изображение выглядит как текст

Автоматически созданное описание

Для начала, извлечём компонент, показывающий время:

Изображение выглядит как текст

Автоматически созданное описание

Проблема в том, что компонент Clock не обновляет себя каждую секунду автоматически. Для этого добавим так называемое «состояние» (state) в компонент Clock. «Состояние» очень похоже на уже знакомые нам пропсы, отличие в том, что состояние контролируется и доступно только конкретному компоненту.

Преобразование функционального компонента в классовый:

Изображение выглядит как текст

Автоматически созданное описание 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.

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

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

Каждый раз когда DOM-узел, созданный компонентом, удаляется, происходит «размонтирование» (unmounting).

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

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

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

Метод componentDidMount() запускается после того, как компонент отрендерился в DOM.

Метод componentWillMount() сбрасывает таймер в методе жизненного цикла.

Реализуем метод tick(). Он запускается таймером каждую секунду и вызывает this.setState() (планирует обновление внутреннего состояния компонента):

Изображение выглядит как текст

Автоматически созданное описание

Важно знать три детали о правильном применении setState().

1.Нельзя изменять состояние напрямую:

Изображение выглядит как текст

Автоматически созданное описание Изображение выглядит как текст

Автоматически созданное описание

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

2. 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.

Изображение выглядит как текст

Автоматически созданное описание Изображение выглядит как текст

Автоматически созданное описание

Правильно будет использовать второй 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.

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.

Этот процесс называется «нисходящим» («top-down») или «однонаправленным» («unidirectional») потоком данных. Состояние всегда принадлежит определённому компоненту, а любые производные этого состояния могут влиять только на компоненты, находящиеся «ниже» в дереве компонентов.

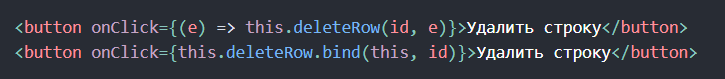
**Обработка событий:**В React нельзя предотвратить обработчик события по умолчанию, вернув false. Нужно явно вызвать preventDefault:

Изображение выглядит как текст, экран, снимок экрана, серебряный

Автоматически созданное описание

При использовании React обычно не нужно вызывать addEventListener, чтобы добавить обработчики в DOM-элемент после его создания. Вместо этого добавьте обработчик сразу после того, как элемент отрендерился.

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



**Условный рендеринг:**

Рассмотрим два компонента:

Изображение выглядит как текст

Автоматически созданное описание

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

Изображение выглядит как текст

Автоматически созданное описание

Вы можете [внедрить любое выражение в JSX](https://ru.reactjs.org/docs/introducing-jsx.html#embedding-expressions-in-jsx), заключив его в фигурные скобки. Это правило распространяется и на логический оператор && языка JavaScript, которым можно удобно вставить элемент в зависимости от условия.

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.

**Списки и ключи:**

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

Изображение выглядит как текст

Автоматически созданное описание

Ключи нужно определять непосредственно внутри массивов.

Например, если вы [извлекаете](https://ru.reactjs.org/docs/components-and-props.html#extracting-components) компонент ListItem, то нужно указывать ключ для <ListItem /> в массиве, а не в элементе <li> внутри самого ListItem.

Неправильное использование ключей:

Изображение выглядит как текст

Автоматически созданное описание

Правильное использование ключей:

Изображение выглядит как текст

Автоматически созданное описание

Как правило, элементам внутри map() нужны ключи.

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.

**Формы:**

По умолчанию браузер переходит на другую страницу при отправке HTML-форм. Однако, чаще всего форму удобнее обрабатывать с помощью JavaScript-функции, у которой есть доступ к введённым данным. Стандартный способ реализации такого поведения называется «управляемые компоненты».

Допустим, мы хотим, чтобы предыдущий пример выводил на экран имя, когда мы отправляем форму. Тогда можно написать форму в виде управляемого компонента:

Изображение выглядит как текст

Автоматически созданное описание

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.

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

Изображение выглядит как текст

Автоматически созданное описание

**Подъём состояния(дописать пример):**

Часто несколько компонентов должны отражать одни и те же изменяющиеся данные.

В React совместное использование состояния достигается перемещением его до ближайшего предка компонентов, которым оно требуется. Это называется «подъём состояния».

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:**

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

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

Изображение выглядит как текст

Автоматически созданное описание

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

Изображение выглядит как текст

Автоматически созданное описание

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.

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:

Изображение выглядит как текст

Автоматически созданное описание

**Хуки:**

**Хуки позволяют вам повторно использовать логику состояния, не затрагивая дерево компонентов.**

**Хуки позволяют разбить один компонент на маленькие функции по их назначению (например, подписке или загрузке данных).**