# React

**React** — это JavaScript-библиотека для разработки пользовательского интерфейса. (декларативная)

**Основные характеристики:**

* **Компонентный подход.** Пользовательский интерфейс разбивается на небольшие, независимые и переиспользуемые компоненты.
* **Виртуальный DOM,** который используется для эффективного обновления только измененных частей интерфейса. Каждый раз, когда происходят изменения, React сравнивает виртуальный DOM с реальным, и обновляет только необходимые элементы.
* **Однонаправленный поток данных.** Данные передаются от родительских компонентов дочерним. Это обеспечивает предсказуемость и упрощает управление состоянием приложения.
* **JSX (JavaScript XML)** – расширение синтаксиса JavaScript, которое позволяет описывать структуру пользовательского интерфейса в виде XML-подобного кода прямо внутри JS. Это облегчает интеграцию HTML и JavaScript.
* **Реактивные обновления**: React обеспечивает автоматическое обновление пользовательского интерфейса при изменении состояния.

# Функции создания компонентов

## React.createElement( type, [props], [...children])

создает и возвращает новый React-элемент определенного типа. Аргументом type может быть строка, содержащая имя тега (например, ‘div’ или ‘span’), React-компонент (класс или функция), или React-фрагмент.

Код, написанный с использованием JSX, будет преобразован в React.createElement(). Обычно вы не будете вызывать React.createElement() напрямую, если вы используете JSX.

## ReactDOM.render(element, container[, callback])

устаревший метод, который рендерит указанный элемент в указанный контейнер DOM и возвращает ссылку на компонент.

Если элемент уже был ранее отрендерен в container, то повторный вызов произведет его обновление и изменит соответствующую часть DOM, чтобы она содержала последние изменения.

Callback вызывается после того, как компонент отрендерится.

## ReactDOM.createRoot()

это новый метод, добавленный в React версии 16.9, который используется для создания корневого компонента React, вместо render().



# JSX

**JSX -** синтаксический сахар для функции React.createElement(component, props, ...children).



Скомпилируется в



Значения *false*, *null*, *undefined* и *true* — валидные дочерние компоненты. Просто они не рендерятся.

**JSX** — это расширение синтаксиса для JavaScript, позволяющее писать HTML-подобную разметку внутри файла JavaScript.

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

За правильный парсинг и дальнейшую обработку отвечает **babel –** специальный компилятор, который переводит jsx в js**.** Он также используется для перевода современного js для запуска в старых браузерах.

## Правила JSX

### Вернуть один корневой элемент

Чтобы вернуть несколько элементов из компонента, оберните их одним родительским тегом.

можете использовать <div>:

Если не хотите добавлять лишний <div>в свою разметку, вы можете вместо этого написать <>and </>

### Закройте все теги

JSX требует, чтобы теги были закрыты явно: самозакрывающиеся теги, такие как, <img>должны стать <img />, а теги-обертки, такие как, <li>orangesдолжны быть записаны как <li>oranges</li>

### CamelCase самое необходимое!

**CamelCase** (с англ. — «ВерблюжийРегистр», также «ГорбатыйРегистр», «СтильВерблюда») — стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово внутри фразы пишется с прописной буквы. Стиль получил название CamelCase, поскольку прописные буквы внутри слова напоминают горбы верблюда (англ. Camel).

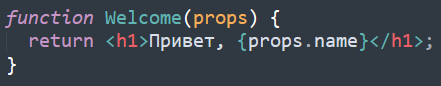
JSX превращается в JavaScript, а атрибуты, написанные в JSX, становятся ключами объектов JavaScript. В ваших собственных компонентах вам часто потребуется считывать эти атрибуты в переменные. Но JavaScript имеет ограничения на имена переменных. Например, их имена не могут содержать тире или быть зарезервированными словами, такими как class.

Вот почему в React многие атрибуты HTML и SVG записываются в camelCase. Например, вместо stroke-widthвы используете strokeWidth. Поскольку class это зарезервированное слово, в React **className** вместо этого.

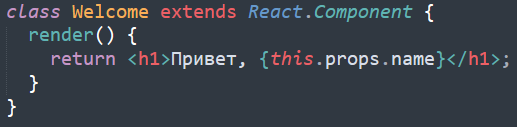
# Компоненты, виды компонентов

**React-компоненты –** повторно используемые части кода, которые возвращают элементы для отображения на странице.

## Функциональные компоненты



## Классовые компоненты



Классовые компоненты наследуются от класса **React.Component**. Он имеет единственный обязательный метод **render().**

## Управляемыми и неуправляемыми компоненты

Элемент называется **управляемым**, если его значение контролирует React.

HTML элементы (input, textarea, select) - сами управляют своим состоянием и сами обновляются когда пользователь вводит данные. Это **неуправляемые** компоненты - они хранят свои данные прямо в DOM.

Чтобы прочитать их значения, используются рефы.

Для того чтобы сделать управляемый компонент, нужно чтобы значение этих элементов хранилось именно в состоянии React компонента, т.е. управлялось через setState/useState, таким образом состояние React-компонента становится единственным источником правды для этих элементов.

Обычно компонент с некоторым локальным состоянием(state) называют «**неконтролируемым**».

«**Контролируется**», когда важная информация в нем управляется свойствами(props), а не его собственным локальным состоянием(state). Это позволяет родительскому компоненту полностью определять поведение.

На практике «управляемый» и «неуправляемый» не являются строгими техническими терминами — каждый компонент обычно имеет некоторое сочетание локального состояния и свойств.

При написании компонента учитывайте,

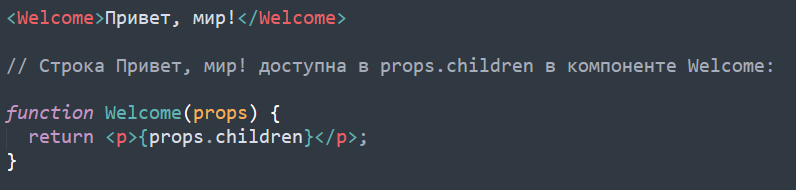
* какая информация в нем должна контролироваться (через props),
* какая информация не должна контролироваться (через state).

# Props

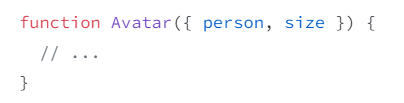
**Props (сокращение от "properties") –** это входные данные для React-компонентов, передаваемые от родительского компонента дочернему компоненту.

Props предназначены **только для чтения**. Ни в каком случае их не следует изменять. Если вам нужно поменять значение в ответ на пользовательский ввод или ответ сервера, используйте *state* (состояние).

**props.children –** контент между открывающимся и закрывающимся тегом компонента. Например:



Мы также можем «деструкторизировать» пропсы чтобы обращаться к ним напрямую, а не через объект props.



# State

**State –** объект который хранит информацию о состоянии компонента.

Обычные локальные переменные компонента не сохраняются между рендерами. Когда React рендерит компонент, он рендерит его с нуля, не учитывая никаких изменений локальных переменных. Изменение локальных переменных также не вызывают рендеринга. React не понимает, что ему нужно отрисовать компонент заново с новыми данными.

Доступ к State имеет только сам компонент. Состояние компонента будет недоступно из вне, например из родительского компонента.

## Изменений стэйта

Изменение состояния происходит не напрямую, а через вызов метода **setState()**. Когда state изменяется, React перерисовывает компонент, чтобы отразить новое состояние.

Например, компоненту Checkbox может понадобиться состояние isChecked, а компоненту NewsFeed необходимо отслеживать посты при помощи состояния fetchedPosts.

В **setState можно передать колбэк**, который выполнится после обновления стейта. Так как setState асинхронный, его значение не изменяется сразу же. Если вам нужно использовать обновленный стейт для выполнения каких-либо действий либо AJAX запросов, можете передать колбэк.

Пример компонента счетчика с меняющимся состоянием:



# Методы жизненного цикла компонента

## Монтирование

При создании экземпляра компонента и его вставке в DOM, следующие методы вызываются в установленном порядке:

**constructor(), constructor(props)**

Вы можете не использовать конструктор в React-компоненте, если вы не определяете состояние или не привязываете методы.

Конструктор вызывается до того, как компонент будет примонтирован. В начале конструктора необходимо вызывать **super(props)**. Если этого не делать this.props не будет определен.

Конструкторы обычно используются для двух целей:

* Инициализация внутреннего состояния через присвоение объекта this.state.
* Привязка обработчиков событий к экземпляру.

Вы не должны вызывать setState() в конструкторе. Начальное состояние можно присвоить напрямую в **this.state.** Конструктор, единственное место, где можно напрямую изменить this.state. В остальных методах необходимо использовать this.setState().

Не используйте побочные эффекты или подписки в конструкторе. Вместо этого используйте *componentDidMount()*.

**static getDerivedStateFromProps() –** используется для обновления state на основе props. Вызывается перед каждым рендерингом компонента, как нового, так и при обновлении. Этот метод существует для редких случаев, когда состояние зависит от изменений в пропсах.

Принимает 2 параметра: props и state. Он должен вернуть объект, представляющий новое состояние объекта, или null если состояние не требуется обновлять.

**render() –** обязательный метод компонента React, который определяет, что будет отображаться на экране. Он возвращает React-элементы (JSX).

Метод *render()* вызывается автоматически при каждом обновлении компонента или его родительского компонента. Он должен быть чистой функцией без побочных эффектов, то есть он не должен изменять состояние компонента и не должен взаимодействовать напрямую с брузером.

Взаимодействовать с браузером необходимо в *componentDidMount()* или других методах жизненного цикла. Чистый render() делает компонент понятным.

*Render()* не вызывается, если *shouldComponentUpdate()* возвращает false.

render() может возвращать один из следующих вариантов:

* **Элемент React.** Обычно создается с помощью JSX.
* **Массив.** Возвращает несколько элементов из render().
* **Портал.** Позволяет рендерить дочерние элементы в DOM-узлы, которые находятся вне иерархии компонентов.

Порталы полезны когда вам нужно отобразить содержимое вне корневого узла вашего приложения, например, для создания модальных окон, всплывающих подсказок и т.п.

* **Строки и числа –** рендерит текстовые DOM-узлы.
* **Boolean или null –** ничего не рендерит. Обычно необходим для поддержки паттерна *return test && <Child />*, где test — логическое значение)

**componentDidMount()** вызывается сразу после монтирования (то есть вставки компонента в DOM). В этом методе должны происходить действия, которые требуют наличия DOM-узлов. Это хорошее место для создания сетевых запросов.

Этот метод подходит для настройки подписок. Но не забудьте отписаться от них в *componentWillUnmount()*.

## Обновление

Обновление происходит при изменении пропсов или состояния. Следующие методы вызываются в установленном порядке при повторном рендере компонента.

static **getDerivedStateFromProps()** – обновление стэйта который зависит от пропсов до render().

**shouldComponentUpdate()** – вызывается перед обновлением компонента и позволяет контролировать, должен ли компонент перерисовываться или нет.

В теле метода `shouldComponentUpdate()` вы можете реализовать свою логику сравнения текущих свойств и состояния компонента с новыми `nextProps` и `nextState`. Верните true чтобы отрендерить компонент, и false чтобы отменить рендер.

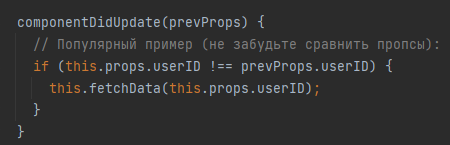
По умолчанию происходит повторный рендер при любом изменении состояния или пропсов. В большинство случаев вы должны полагаться на это поведение.

**render() –** аналогичен как при монтировании.

**getSnapshotBeforeUpdate** вызывается прямо перед этапом “фиксирования” (например, перед добавлением в DOM). Он позволяет вашему компоненту брать некоторую информацию из DOM перед её возможным изменением. Любое значение, возвращаемое этим методом жизненного цикла, будет передано как третий параметр *componentDidUpdate()*.

**componentDidUpdate() –** вызывается сразу после обновления. Не вызывается при первом рендере.

Метод позволяет работать с DOM при обновлении компонента. Также он подходит для выполнения таких сетевых запросов, которые выполняются на основании результата сравнения текущих пропсов с предыдущими. Если пропсы не изменились, новый запрос может и не требоваться.



В componentDidUpdate() можно вызывать setState(), однако его **необходимо обернуть в условие**, как в примере выше, чтобы не возник бесконечный цикл. Вызов setState() влечет за собой дополнительный рендер, который незаметен для пользователя, но может повлиять на производительность компонента. Вместо «отражения» пропсов в состоянии рекомендуется использовать пропсы напрямую. Это уменьшит потребление памяти, поможет избежать избыточных обновлений и перерисовок, избавит от несогласованности данных (при обновлении пропсов забыли обновить state).

Примечание:

**Эти методы устарели**. Не используйте их в новом коде.

**UNSAFE\_componentWillUpdate()**

**UNSAFE\_componentWillReceiveProps()**

## Размонтирование

Этот метод вызывается при удалении компонента из DOM:

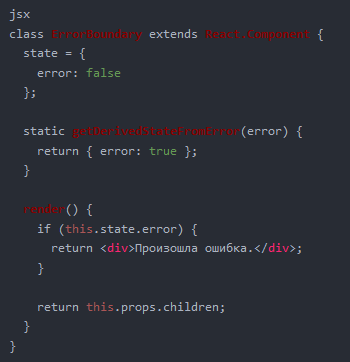
**componentWillUnmount()** - вызывается непосредственно перед размонтированием и удалением компонента. В этом методе выполняется необходимый сброс: отмена таймеров, сетевых запросов и подписок, созданных в componentDidMount().

В методе `componentWillUnmount()` нельзя вызывать метод `setState()`, потому что компонент уже будет удален из DOM и его состояние больше не существует. Вызов `setState()` после `componentWillUnmount()` не имеет смысла и может привести к ошибкам или неопределенному поведению.

## Обработка ошибок

Следующие методы вызываются, если произошла ошибка в процессе рендеринга, методе жизненного цикла или конструкторе любого дочернего компонента.

static **getDerivedStateFromError(error) -** используется для обработки ошибок, возникающих в дочерних компонентах и обновления состояния родительского компонента на основе этих ошибок. Он не вызывается при возникновении ошибки в самом компоненте или его методах. Этот метод должен вернуть новое состояние родительского компонента или null если не требуется менять состояние. Можно использовать для рендеринга запасного UI в случае отлова ошибки.



**componentDidCatch() -** позволяет компоненту перехватывать ошибки, которые не были обработаны внутри дочерних компонентов, и выполнять определенные действия в ответ на эти ошибки. Этот метод не возвращает значение. Можно использовать для логирования информации об отловленной ошибке.

## PureComponent

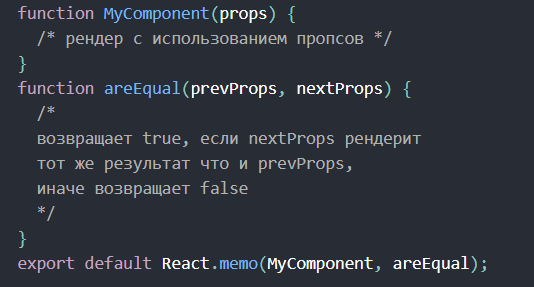
Класс аналогичен **Component.** Отличие заключается в том, что React.Component не реализует **shouldComponentUpdate()**, а **React.PureComponent** реализует его поверхностным сравнением (не обрабатывает вложенные объекты а сравнивает через ===) пропсов и состояния.

Если метод render() вашего React-компонента всегда рендерит одинаковый результат при одних и тех же пропсах и состояниях, для повышения производительности в некоторых случаях вы можете использовать React.PureComponent.

**React.memo**

Если ваш компонент рендерит всегда одно и то же при неменяющихся пропсах, вы можете обернуть его в вызов React.memo. В этом случае React будет использовать результат последнего рендера, избегая повторного рендеринга. React.memo затрагивает **только изменения пропсов.**

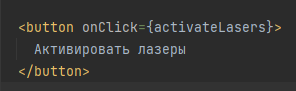
По умолчанию он поверхностно сравнивает пропсы. Можно передать свою функцию сравнения в качестве второго аргумента.



Этот метод предназначен только для оптимизации производительности. Не полагайтесь на него, чтобы предотвратить рендер. Это может привести к ошибкам.

# Обработка событий

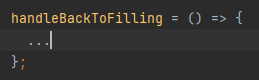
Обработка событий в React-элементах очень похожа на обработку событий в DOM-элементах.



В React нельзя предотвратить обработчик события по умолчанию, вернув false. Нужно явно вызывать **preventDefault**. Например по умолчанию при сабмите формы браузер перезагружает страницу.

В классах же в качестве обработчика выступает один из методов класса. Однако передача его в качестве обработчика обязует привязать к методу bind или использовать стрелочные функции. Но если использовать стрелочные функции, то каждый раз будет создаваться новая функция при каждом рендере.

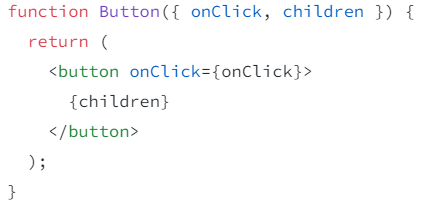
**Стрелочная функция обработчик**



Типы событий, на которые можно повесить обработчик:

* onClick
* onMouseEnter

Обработчики событий можно передавать через пропсы.



Также как и в JS события всплывают от нижнего элемента к верхнему. В обработчике мы можем получить доступ к объекту события и остановить всплытие вызвав метод **stopPropagation()**.



При погружении событие вызывает обработчики **OnClickCapture.** Затем вызывается *onClick* на target-объекте (по которому кликнули). И затем событие всплывает вызывая обработчики *onClick* родительских элементов.

# Рендеринг списка элементов

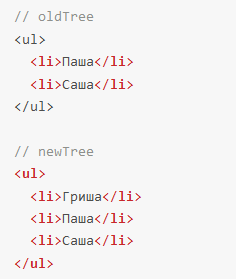
Вы можете создать коллекцию элементов и встроить её в JSX с помощью фигурных скобок {}. Как правило, вы будете рендерить списки внутри какого-нибудь компонента.

При рендере списка необходимо использовать *ключи (key)*.

Атрибут **key** используется для уникальной идентификации элементов в списке компонентов. Ключи помогают React оптимизировать процесс обновления компонентов и эффективно отслеживать изменения в списке.

Ключи должны быть уникальными в пределах списка или коллекции элементов. Рекомендуется использовать стабильные идентификаторы, такие как ID из БД.

В качестве ключа не рекомендуется использовать индекс в массиве. Особенно если порядок элементов может измениться или элементы могут быть добавлены или удалены из середины списка.



При добавлении элемента в начало Реакт сравнит <li>Паша</li> с <li>Гриша</li>  —  обновит его. Затем сравнит <li>Саша</li> с <li>Паша</li>  —  обновит его и в конце создаст <li>Саша</li>. При вставке элемента в начало реакт обновит все элементы в массиве.

При добавлении key реакт будет сравнивать элементы не друг за другом, а будет искать по значению ключа.



Реакт найдет key='1', key='2', определит, что c ними не произошло изменений, и затем найдет новый элемент <li key=’3'>Гриша</li> и добавит только его. Следовательно, с ключами реакт обновит только один компонент.

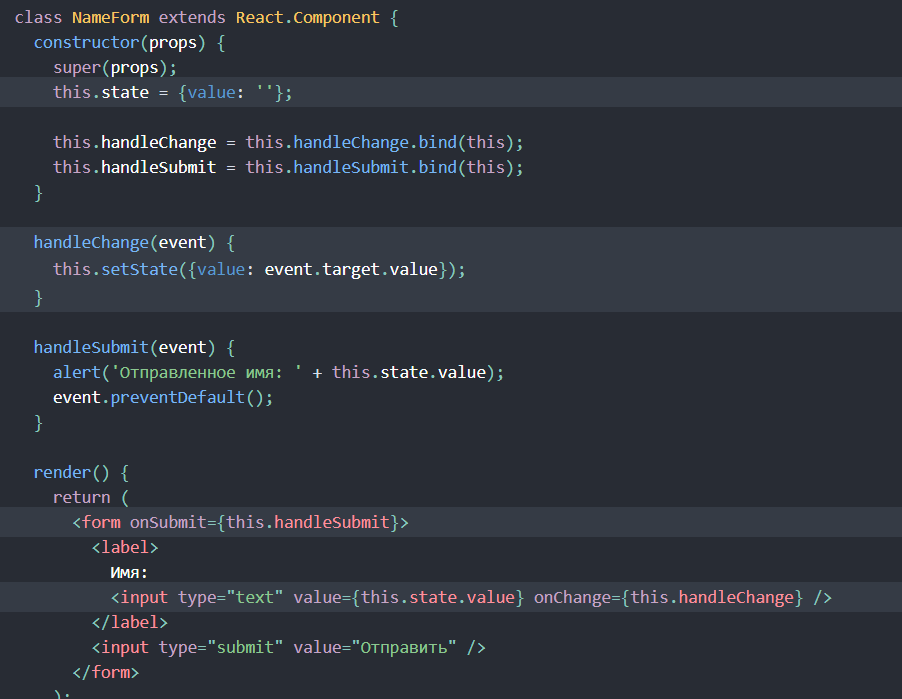
# Работа с формами

Вы можете использовать стандартный тег **form** чтобы создать форму. По умолчанию браузер переходит на другую страницу (или перезагружает текущую) при отправке форм. Однако форму чаще всего удобнее обрабатывать с помощью JS. Стандартный способ реализации такого поведения называется **управляемые компоненты.**

В HTML элементы формы (<input>, <textarea> и <select>) сами управляют своим состоянием. В React мутабельное состояние обычно содержится в state.

В **управляемом компоненте** значение поля ввода всегда определяется состоянием React. Таким образом мы можем контролировать компонент (например ограничивать ввод, проводить валидацию на ходу а не после нажатия кнопки submit).

Обычно на каждый onChange мы обновляем значение поля формы, и выводим новое значение на экран. Таким образом это значение мы сможем передавать другим обработчикам, либо отправлять на сервер.

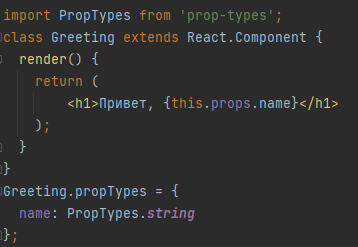


В качестве альтернативы можно использовать **неуправляемые компоненты**. Они хранят данные формы прямо в DOM. Прочитать значения из них мы можем используя ref.

# Проверка типов пропсов с помощью встроенного механизма

React предоставляет встроенные возможности для проверки типов, даже если вы не используете TypeScript.

Для запуска этой проверки вам нужно использовать специальное свойство **propTypes**.



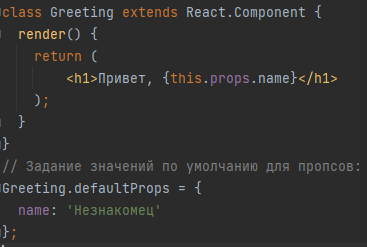
**PropTypes** предоставляет ряд валидаторов, которые могут использоваться для проверки, что получаемые данные корректны. Когда какой-то проп имеет некорректное значение, в консоли будет выведено предупреждение. По соображениям производительности propTypes проверяются только в режиме разработки.

Возможные значения PropTypes:

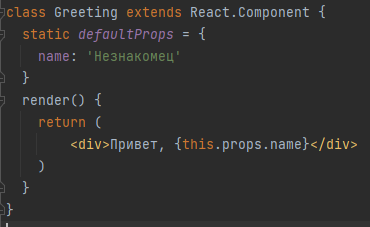
* **array**
* **bool**
* **func**
* **number**
* **string**
* **symbol**
* **node**
* **element** – React элемент
* **elementType** – тип React-элемента (например, MyComponent)
* **instanceOf**(Message) – проп должен быть экземпляром класса.
* **oneOf**(['News', 'Photos']) – одно из указанных значений
* **oneOfType**([PropTypes.string, PropTypes.number, PropTypes.instanceOf(Message)]) – один из нескольких типов
* **arrayOf(PropTypes.number) –** массив объектов конкретного типа
* **func.isRequired –** можно добавить isRequired к любому приведенному выше типу, чтобы указать что проп обязательный.

# Значение пропсов по умолчанию

Вы можете задать значения по умолчанию для ваших props с помощью специального свойства **defaultProps:**

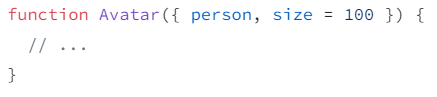


C ES2022 вы можете объявить **defaultProps как статическое свойство** внутри классового React компонента. Для поддержки этого современного синтаксиса в старых браузерах потребуется компиляция.



Определение defaultProps гарантирует, что this.props.name будет иметь значение, даже если оно не было указано родительским компонентом. Сначала применяются значения по умолчанию, заданные в defaultProps. После запускается проверка типов с помощью propTypes. Так что проверка типов распространяется и на значения по умолчанию.

Пропсы по умолчанию в функциональных компонентах можно указать и с помощью синтаксиса деструкторизации:



# Рефы и их назначение

**Refs (ссылки) –** предоставляет способ доступа к DOM-узлам или React-элементам созданным в методе *render()*.

В обычном потоке данных пропсы – единственный способ взаимодействия родительских компонентов со своими дочерними элементами. Чтобы изменить дочерний элемент, вы повторно отрисовываете его с помощью новых свойств. Иногда нужно принудительно модифицировать дочерний элемент за пределами типичного потока данных.

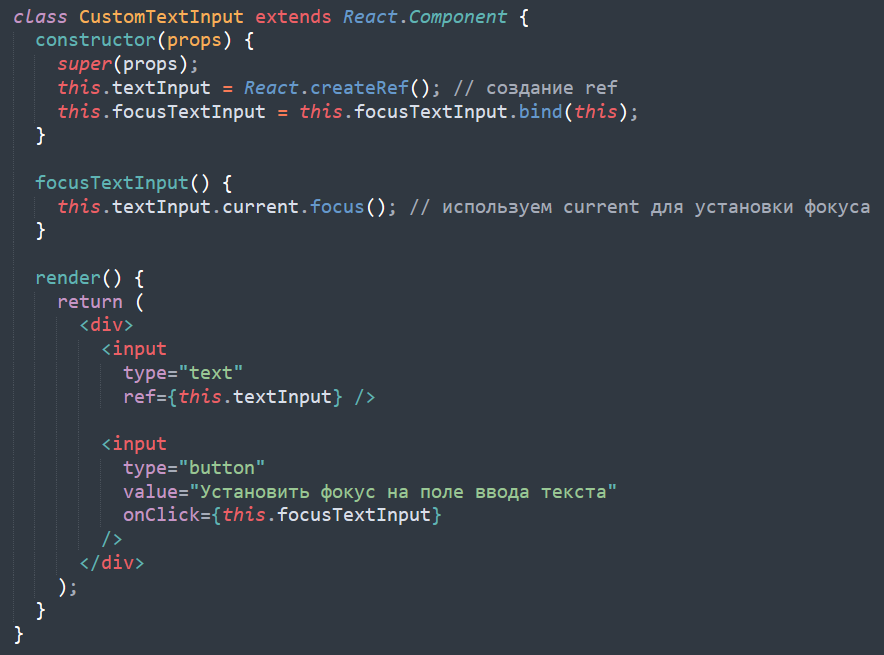
Когда использовать ссылки:

* Управление фокусом, выделение текста или воспроизведение медиаресурсов.
* Выполнение анимаций в императивном подходе.
* Интеграция со сторонними библиотеками, взаимодействующими с DOM.

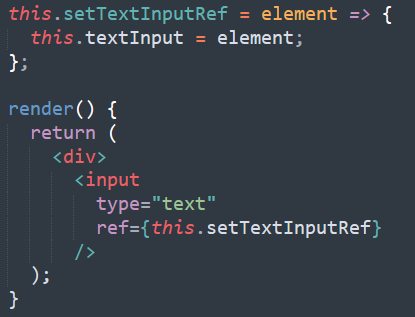
Ссылки создаются с помощью **React.createRef()** и добавляются к компонентам с помощью атрибута **ref**. Ссылки обычно записываются в свойства компонента, чтобы к ним можно было обратиться откуда угодно в компоненте.

**this.myRef.current –** получить доступ к узлу, на который ссылается ссылка.

* Когда ref используется в HTML-элементе, *current* указывает на базовый элемент DOM.
* Когда ref используется на пользовательском классовом компоненте, *current* указывает на примонтированный экземпляр компонента.
* Нельзя создать ref на функциональный компонент, потому что у функционального компонента не может быть экземпляров. Однако мы можем использовать ref внутри функционального компонента, ссылаясь на класс или DOM-элемент.



Вместо использования *createRef()* можно использовать ссылку-callback. Колбэк принимает экземпляр компонента или DOM-элемент, и мы можем его сохранить, либо сделать с ним что-то еще.



Обычно ref используется для прямого обращения к DOM. Следует избегать использования ссылок, особенно для тех вещей, которые могут быть сделаны в декларативном стиле.

Например, вместо использования методов open() и close() в компоненте Dialog, передайте ему свойство isOpen.

# Условный рендеринг

Можем отрендерить по (if, ? :, &&, или вернуть null если нам надо предотвратить рендер компонента).

На практике не распространено возвращение null. Чаще исключают компонент из родителя на основе какого-то условия.

Не ставьте числа слева в выражении && так как если число будет 0, в браузере будет отрисован 0.

# Базовые хуки

**Хуки** – нововведение в React 16.8 которое позволяет использовать состояние и другие возможности React без написания классов. (Раньше у функциональных компонентов не было не состояния, ни методов жизненного цикла, и они были пригодны только для того, чтобы выводить информацию).

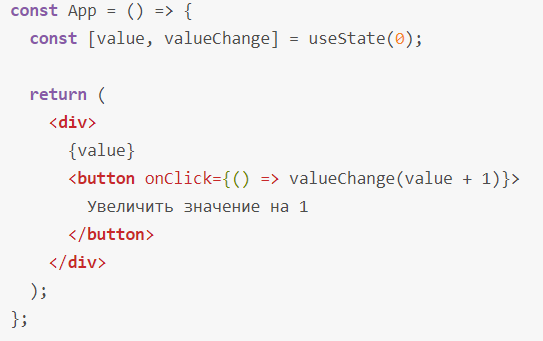
Хуки не работают внутри классов. Хуки используются **вместо** классов.

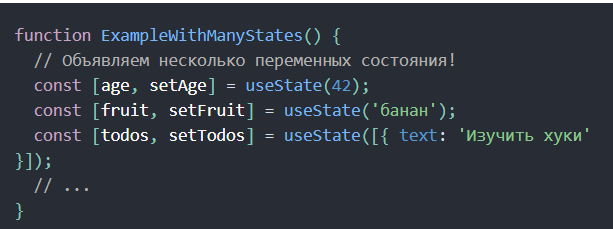
Хуки могут вызываться только на верхнем уровне компонентов. Нельзя вызывать хуки внутри if-ов, циклов или других вложенных функций. Нужно думать о хуках, как о безусловных декларациях о потребностях вашего компонента.

## useState

**useState –** хук для работы с состоянием. Он предоставляет переменную состояния, и функцию для обновления этой переменной и запуска React для повторного рендеринга компонента.

Хук useState принимает в качестве параметра начальное значение. В отличии от классовых компонентов, состояние может и не быть объектом.

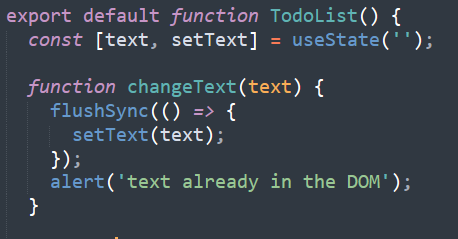
****



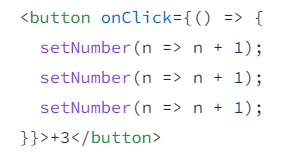
В классовом компоненте мы можем создать только одно общее состояние компонента, а в функциональном несколько. И все они будут независимы друг от друга, но каждое из них будет вызывать рендеринг компонентов.

Также в отличии от this.setState обновление переменной состояния всегда замещает ее значение, а не осуществляет слияние.

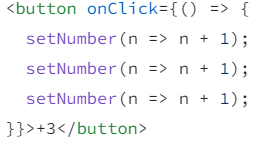
Состояние обновляется асинхронно. Если нам нужно, чтобы дом обновлялся синхронно, мы можем обернуть сеттер состояния в **flushSync.**



Из-за того, что состояние обновляется асинхронно, не будет работать и такой код:



Значение увеличится только на 1. Если вы хотите **обновить переменную состояния несколько раз** перед рендерингом, можно передавать функцию, вместо конкретного значения состояния:



React ставит эту функцию в очередь на обработку после выполнения всего остального кода в обработчике события. Во время следующего рендеринга, React выполняет все эти функции и выдает окончательное обновленное состояние.

## useRef

useRef - это хук React, позволяющий ссылаться на значение, которое не нужно для рендеринга, но которое нужно сохранять между повторными рендерами.

import { useRef } from 'react';  
  
function MyComponent() {  
const intervalRef = useRef(0);  
const inputRef = useRef(null);  
// ...  
}

* **initialValue**: Значение, которое вы хотите, чтобы свойство current объекта ref было первоначальным. Это может быть значение любого типа. Этот аргумент игнорируется после первоначального рендеринга.

Функция useRef возвращает объект с одним свойством:

* current: Изначально установлено initialValue. Позже вы можете установить его в другое значение. Если вы передадите объект ref в React как атрибут ref узла JSX, React установит его свойство current.

Используя ссылку, вы гарантируете, что:

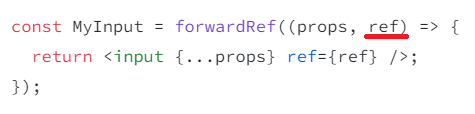
* Вы можете **сохранять информацию** между повторными рендерами (в отличие от обычных переменных, которые сбрасываются при каждом рендере).
* Ее изменение **не вызывает повторного рендеринга** (в отличие от переменных состояния, которые вызывают повторный рендеринг).
* Информация **является локальной** для каждой копии вашего компонента (в отличие от внешних переменных, которые являются общими).

Наиболее часто ref используется для доступа к элементу DOM. Например, для установки фокуса, скрола к элементу, изменения его размера и положения.

Если нам нужно иметь динамическое количество ссылок, можно использовать **ref callback.** В атрибут ref в jsx мы передаем колбэк, который будет получать каждый узел, и мы сможем записать его например в массив и затем использовать массив рефов.



По умолчанию нельзя создать ссылку на дочерний React-компонент. Мы должны явно указать в дочернем компоненте, что принимаем ref.



Не стоит обращаться к ref во время рендеринга. Во время первого рендеринга узлы DOM еще не будут созданы, поэтому *ref.current* будет равен null. А во время рендеринга обновлений узлы DOM еще не были обновлены. Поэтому читать их еще рано.

Обычно доступ к ref осуществляется из обработчиков событий. Если нет подходящего события, вам может понадобится Эффект.

## useEffect

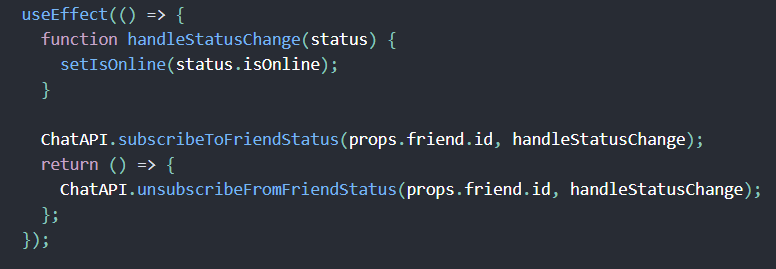
**useEffect –** дает вам возможность выполнять побочные эффекты в функциональном компоненте, такие как загрузка данных, оформление подписки и изменение DOM вручную. Представляет собой совокупность методов *componentDidMount*, *componentDidUpdate*, и *componentWillUnmount*.

React запомнит функцию (эффект) которую вы передали и вызовет ее после того, как внесет все изменения в DOM. Вторым параметром useEffect принимает массив аргументов.

useEffect выполняется после каждого рендера и обновления, а также при изменении переданных ему аргументов.



Чтобы отвязать обработчик событий (что в классовом компоненте делается внутри componentWillUnmount) мы возвращаем callback. Однако React будет вызывать его при каждом рендере компонента, что может сказаться на производительности. Используя второй параметр хука useState мы можем указать аргументы. Эффект будет перезапускаться только при изменении состояния этих аргументов, таким образом мы можем уменьшить количество вызовов эфекта.



В некоторых эффектах нет этапа сброса, поэтому они не возвращают ничего.

Если вы хотите запустить эффект и сбросить его только один раз (при монтировании и размонтировании), вы можете **передать пустой массив** ([]) вторым аргументом. React посчитает, что ваш эффект не зависит от каких-либо значений из пропсов или состояния и поэтому не будет выполнять повторных запусков эффекта.

## useMemo

**useMemo() –** хук, который используется для оптимизации производительности компонентов путем кеширования результата вычислений. Результат берется из кэша, если не изменились значения отслеживаемых пропсов.

