**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 юыть 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.

**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.

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

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



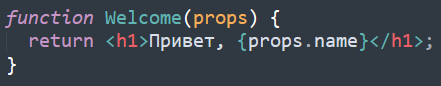
Скомпилируется в



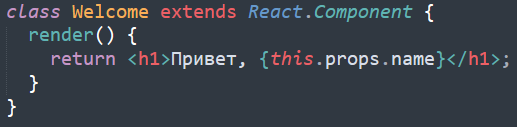
Значения *false*, *null*, *undefined* и *true* — валидные дочерние компоненты. Просто они не рендерятся.

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

Компонент может быть **Функциональным**:



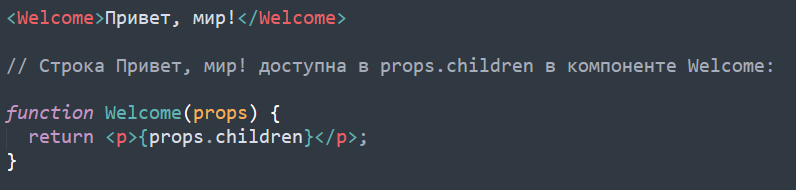
Либо **классовый**:



**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. Нпример:



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

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.

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.

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.

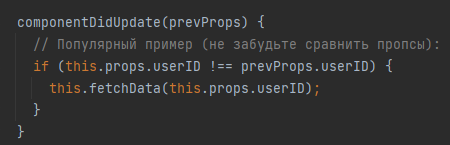
По умолчанию происходит повторный рендер при любом изменении состояния или пропсов. В большинство случаев вы должны полагаться на это поведение.

**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 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:

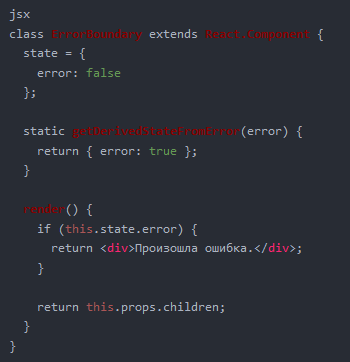
**componentWillUnmount()** - вызывается непосредственно перед размонтированием и удалением компонента. В этом методе выполняется необходимый сброс: отмена таймеров, сетевых запросов и подписок, созданных в componentDidMount().

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.

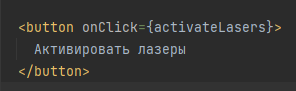
static **getDerivedStateFromError(error) -** используется для обработки ошибок, возникающих в дочерних компонентах и обновления состояния родительского компонента на основе этих ошибок. Он не вызывается при возникновении ошибки в самом компоненте или его методах. Этот метод должен вернуть новое состояние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



**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.

**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.

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.

**Рендеринг списка элементов**

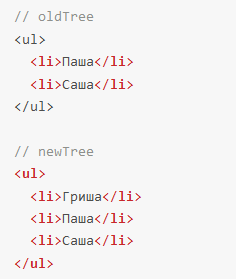
Вы можете создать коллекцию элементов и встроить её в JSX с помощью фигурных скобок {}. Как правило, вы будете рендерить списки внутри какого-нибудь компонента.

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.



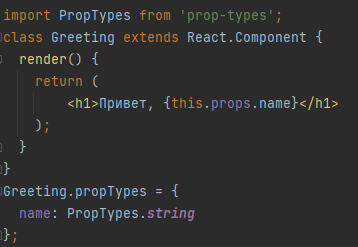
Реакт найдет key='1', key='2', определит, что c ними не произошло изменений, и затем найдет новый элемент <li key=’3'>Гриша</li> и добавит только его. Следовательно, с ключами реакт обновит только один компонент.

**Работа с формами**

**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**.

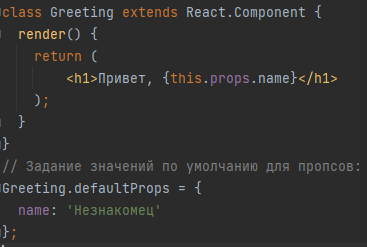


**PropTypes** предоставляет ряд валидаторов, которые могут использоваться для проверки, что получаемые данные корректны. Когда какой-то проп имеет некорректное значение, в консоли будет выведено предупреждение. По соображениям производительности propTypes проверяются только в режиме разработки.

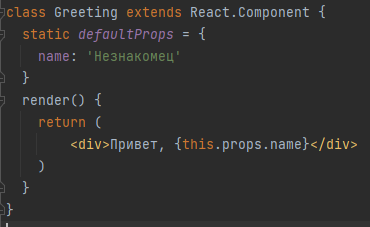
Возможные значения PropTypes:

* **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.

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.

**Рефы и их назначение**

**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**

**useState**

**useRef**

**useEffect**