**Лекция 2.**
**Повторно используемые компоненты и потоки данных**

Изучите передачу данных между компонентами и управление выводом пользовательского интерфейса. Сосредоточьтесь на свойствах (props), условном рендеринге и рендеринге списков данных.

## 1. Передача данных с помощью свойств (props)

Компонентам часто необходимо отображать разные данные в зависимости от места их использования. В React данные передаются в компоненты с помощью **props** . Свойства (props) — это входные параметры компонента. Они позволяют передавать данные из **родительского компонента** в **дочерний** и управлять тем, что отображает компонент.

Вместо того чтобы жестко задавать значения внутри компонента, **свойства (props) делают компоненты многократно используемыми** . Один и тот же компонент можно использовать несколько раз с разными данными, создавая разный пользовательский интерфейс.

Ниже приведён простой пример передачи данных компоненту с помощью свойств (props):

```
function Greeting(props) {
  return <h2>Hello, {props.name}</h2>;
}

function App() {
  return <Greeting name="Olivia" />;
}
```

В этом примере **App** компонент передает значение "**Olivia**" другому **Greeting** компоненту, используя свойство с именем **name. Greeting** компонент получает эти данные и отображает их в пользовательском интерфейсе.

**Примечание.Свойства (props) доступны только для чтения .** Компонент не должен изменять получаемые им свойства. Их цель – управлять внешним видом и поведением компонента на основе внешних данных.

## 2. Условная отрисовка в React

Условный рендеринг позволяет динамически отображать или скрывать элементы в зависимости от определенных условий, повышая гибкость компонентов React. В этом разделе мы рассмотрим два стандартных метода условного рендеринга: **оператор**&& и **тернарный оператор**.

### 2.1. Условная отрисовка с использованием оператора &&

Синтаксис. Оператор &&`<react>` в React используется для условного рендеринга и работает аналогично оператору if`<recall>` в JavaScript. Он позволяет отображать элементы в зависимости от заданных условий.

```
condition && (<p>element</p>)
```

Этот метод часто используется, когда необходимо отобразить элемент только в том случае, если выполняется определенное условие.

**Пример**. Рассмотрим пример, когда мы хотим уведомить студентов, успешно сдавших экзамен. Если оценка студента превышает 60 баллов, мы отобразим сообщение об успешной сдаче экзамена с указанием его имени и оценки.

```
const Notification = (props) =>
  props.mark > 60 && (
    <p>
      {props.name} has passed the test with {props.mark} points.
    </p>
  );
```

Компонент **Notification** отображает элемент абзаца **<p>** в зависимости от **mark** значения свойства.

Результат **Эмили** не отображается, потому что он меньше **60pts**.

### 2.2. Условная отрисовка с использованием тернарного оператора

**Синтаксис**. Условная отрисовка с использованием тернарного оператора **(? ... : ...)** — ещё один мощный метод. Он предоставляет лаконичный способ отрисовки элементов в зависимости от условий.

```
condition ? (true_element) : (false_element)
```

Этот подход подходит при выборе между двумя различными элементами на основе определенного условия.

**Пример**. Рассмотрим сценарий, в котором мы хотим по-разному приветствовать пользователей в зависимости от того, вошли ли они в систему. Компонент Greetingдемонстрирует условную отрисовку с помощью тернарного оператора.

```
const Greeting = (props) =>
  props.loggedIn ? (
    <p>Welcome to the home page, {props.name}.</p>
  ) : (
    <p>Hello {props.name}, could you please log in.</p>
```

```
  );
```

В этом примере Greeting компонент приветствует пользователей по-разному в зависимости от loggedInзначения свойства.

**Примечание.** Хотя оба метода позволяют осуществлять условную отрисовку, важно понимать их различия. **Оператор** && идеально подходит, когда необходимо отобразить элемент только при выполнении определенного условия. В отличие от него, **тернарный оператор подходит для случаев ,** когда нужно выбрать один из двух различных элементов на основе условия.

### 3. Отображение списков данных в React

При работе над современными веб-приложениями часто приходится отображать коллекции данных, полученных с бэкэнда. Для этого мы используем метод map(). Внутри этого map()метода мы применяем функцию обратного вызова и возвращаем JSX для отображения желаемого результата.

Давайте рассмотрим пример, чтобы проиллюстрировать, как это работает. У нас будет Арккомпонент, который будет передавать свойство `prop` toys, массив объектов. ToyCardКомпонент будет использовать этот map() метод для отрисовки каждой игрушки в массиве.

```jsx
// Data from backend
const toysData = [
  { id: "id-1", name: "Rainbow Sparkle Unicorn Plush" },
  { id: "id-2", name: "Jungle Adventure Playset" },
  { id: "id-3", name: "Magical Princess Castle Dollhouse" },
  { id: "id-4", name: "RoboBot Transformer Robot" },
  { id: "id-5", name: "SuperBlast Action Figure" },
];

// Child component
const ToyCard = (props) => (
  <ul>
    {props.toys.map((toy) => (
      <li>{toy.name}</li>
    ))}
  </ul>
);

// Parent component
const App = () => (
  <>
    <ToyCard toys={toysData} />
  </>
);
```

В строках 13-15 мы видим использование этого map() метода. Он позволяет нам перебирать каждый элемент массива, извлекать его значение и генерировать определенную разметку на основе данных каждого элемента.

Однако, если мы внимательно посмотрим на консоль, то увидим следующее предупреждение: Each child in a list should have a unique "key" prop..

**Ключ.** Ключевым **параметром** является важная строковая переменная, которую необходимо присвоить при создании элементов внутри коллекции.

React использует ключи для обеспечения стабильной идентификации элементов внутри коллекции. Эти ключи позволяют React определять, какие элементы необходимо перерисовать и пересоздать при изменении, вместо того чтобы пересоздавать всю коллекцию целиком. Использование ключей предотвращает ненужное пересоздание элементов, что приводит к повышению производительности.

**Примечание.** Ключ должен быть **уникальным** среди соседних элементов. Он должен оставаться **неизменным** и не меняться со временем.

Обычно рекомендуется использовать идентификаторы, например, id значения из бэкэнд-данных, в качестве ключевых свойств. Такая практика позволяет React эффективно идентифицировать и управлять отдельными элементами внутри коллекции.

Давайте исправим наше предыдущее приложение, используя ключевые свойства для элементов:

```
// Data from backend
const toysData = [
  { id: "id-1", name: "Rainbow Sparkle Unicorn Plush" },
  { id: "id-2", name: "Jungle Adventure Playset" },
  { id: "id-3", name: "Magical Princess Castle Dollhouse" },
  { id: "id-4", name: "RoboBot Transformer Robot" },
  { id: "id-5", name: "SuperBlast Action Figure" },
];

// Child component
const ToyCard = (props) => (
  <ul>
    {props.toys.map((toy) => (
      <li key={toy.id}>{toy.name}</li>
    ))}
  </ul>
);
// Parent component
const App = () => (
  <>
    <ToyCard toys={toysData} />
  </>
);
```

Строка 14: key свойство устанавливается для элемента, который будет отображаться несколько раз в массиве данных.

**5. Заключение по основам React**

Вы изучили основные компоненты React и то, как компоненты взаимодействуют друг с другом для создания динамических пользовательских интерфейсов.

Вы изучили, как **передавать данные с помощью props** , позволяя родительским компонентам контролировать отображение дочерних компонентов. Это познакомило с идеей **одностороннего потока данных ,** которая является центральной в структуре приложений React.

Вы научились использовать **условный рендеринг** для отображения или скрытия элементов пользовательского интерфейса в зависимости от условий, используя такие шаблоны, как логический **оператор И (**&& ) и **тернарный оператор (** ? ... : ...**)** .

Вы также научились отображать **списки данных** с помощью map()метода `return`, что является распространенным шаблоном при работе с данными из бэкэнда. В процессе вы узнали, почему **свойство `key`** имеет важное значение для эффективного обновления списков в React.

Вместе эти концепции составляют основу практически каждого приложения React. Понимание их значительно упростит создание многократно используемых компонентов, управление логикой пользовательского интерфейса и анализ потоков данных в вашем приложении.

Далее вы выйдете за рамки статических данных и узнаете, как React обрабатывает **состояние и побочные эффекты** , позволяя вашим приложениям реагировать на взаимодействие с пользователем и внешние данные с течением времени.

**Вопросы для самопроверки:**

1. Для чего используются **props** (свойства) в React?
2. Как работает оператор **&&** для условного рендеринга в React?
3. Какой оператор используется для условного рендеринга в React по принципу **if...else**?
4. Какой метод обычно используется для отображения коллекций данных, полученных с бэкенда, в React?
5. Почему важно назначать свойство **key** элементам внутри коллекции в React?
6. Каким должно быть значение свойства **key**, чтобы обеспечить стабильную идентичность элементов внутри коллекции?