4. Хуки useEffect, useRef, useMemo, useCallback

▼ useEffect

Как работает

Позволяет выполнять побочные эффекты (запросы, таймеры, подписки) после рендера компонента или при изменении зависимостей.

Где используется

- Загрузка данных с API при монтировании.
- Подписки/отписки на события.
- Реакция на изменение пропсов или состояния.

Пример

```
import React, { useEffect, useState } from "react";

export function ExampleUseEffect() {
  const [count, setCount] = useState(0);

  // --- Монтирование компонента (пустой массив зависимостей)
  useEffect(() => {
    console.log("Компонент смонтирован");
  }, []);

  // --- При изменении состояния count
  useEffect(() => {
    console.log(`Счётчик изменился: ${count}`);
```

```
  }, [count]);

  return (
    <div>
      <p>Счётчик: {count}</p>
      <button onClick={() => setCount(count + 1)}>Увеличить</button>
    </div>
  );
}
```

▼ useRef

Как работает

Создаёт объект, который хранит значение между рендерами, не вызывая повторный рендер при изменении.

Где используется

- Сохранение DOM-элемента для работы напрямую.
- Хранение любого значения, которое не должно триггерить перендер.

Пример

```
import React, { useRef, useEffect } from "react";

export function ExampleUseRef() {
  const inputRef = useRef(null);

  useEffect(() => {
    // Фокусируем на поле после рендера
    inputRef.current.focus();
  }, []);

  return (
    <div>
```

```
      <input ref={inputRef} placeholder="Фокус автоматически" />
    </div>
  );
}
```

▼ useMemo

Что делает:

Запоминает результат функции (значение), чтобы не пересчитывать его при каждом рендере, если зависимости не изменились.

Простая аналогия:

Представь, что у тебя есть калькулятор, который считает что-то очень долго. Вместо того чтобы считать заново каждый раз, он запоминает последний результат и использует его, если входные данные не поменялись.

Пример

```
import React, { useState, useMemo } from "react";

export function ExampleUseMemo() {
  const [count, setCount] = useState(0);
  const [other, setOther] = useState(false);

  // Тяжелая функция (симулируем задержку)
  const expensiveValue = useMemo(() => {
    console.log("Тяжелое вычисление...");
    let result = 0;
    for (let i = 0; i < 100000000; i++) { result += i; }
    return result + count;
  }, [count]); // пересчитывается только если count изменился

  return (
    <div>
      <p>Результат: {expensiveValue}</p>
      <button onClick={() => setCount(count + 1)}>Изменить count</button>
```

```
      <button onClick={() => setOther(!other)}>Перерисовать без пересчёт
a</button>
    </div>
  );
}
```

Что видно:

- При клике на `Изменить count` – пересчёт запускается.
- При клике на `Перерисовать без пересчёта` – пересчёт не происходит (значение берётся из памяти).

▼ useCallback

Что делает:

Запоминает саму функцию, чтобы она не создавалась заново при каждом рендере.

Почему это нужно:

В React, если функция передаётся в дочерний компонент, она пересоздаётся при каждом рендере. Если дочерний компонент обёрнут в `React.memo`, он всё равно перерендерится, если ссылка на функцию изменилась. `useCallback` сохраняет ссылку на функцию.

Пример

```
import React, { useState, useCallback } from "react";

const Child = React.memo(({ onClick }) => {
  console.log("Рендер Child");
  return <button onClick={onClick}>Увеличить</button>;
});

export function ExampleUseCallback() {
  const [count, setCount] = useState(0);
  const [other, setOther] = useState(false);
```

```
  const increment = useCallback(() => {
    setCount((prev) => prev + 1);
  }, []); // ссылка на функцию не меняется

  return (
    <div>
      <p>Счётчик: {count}</p>
      <Child onClick={increment} />
      <button onClick={() => setOther(!other)}>Перерисовать родителя</button>
    </div>
  );
}
```

Что видно:

- При нажатии на `Перерисовать родителя` **Child** не перерендерится, потому что `increment` остался той же функцией.
- Без `useCallback` `Child` перерисовывался бы каждый раз.