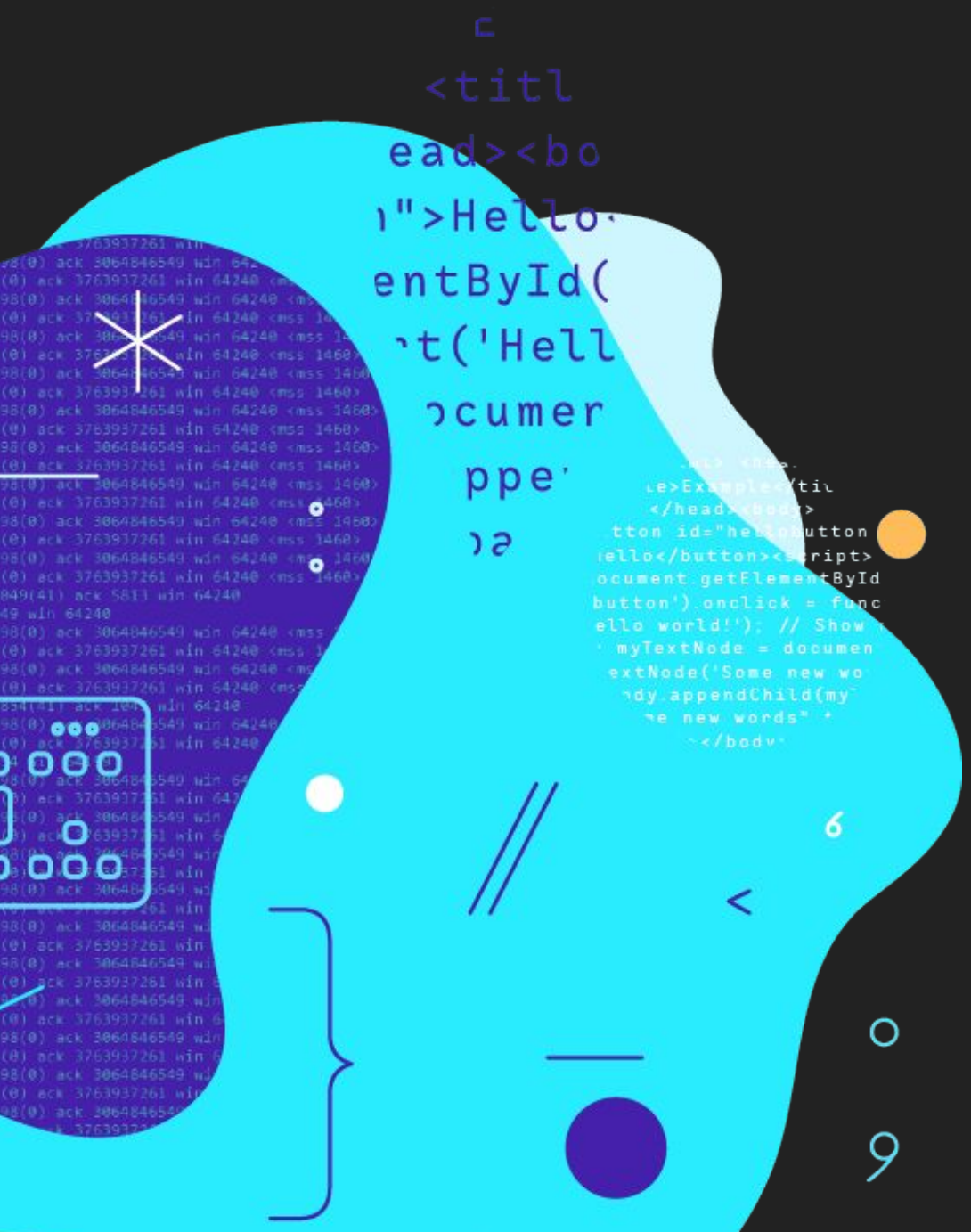

PHASE 3

WEEK 1

DAY 2

ELBRUS/BOOTCAMP

План

1. Custom hooks
2. useContext
3. useReducer
4. memo*, useMemo*, useCallback*

Custom hooks

Custom hooks, пользовательские

Правила хуков:

- Вызывайте хуки только на верхнем уровне.
  НЕ используйте их внутри циклов, условий, вложенных функций.

- Вызывайте хуки только из React-функций.
  Это функциональные компоненты или свои пользовательские хуки.

- Название всех хуков начинается с ‘use...’

Custom hooks, пример

```
import { useEffect, useState } from 'react';

function getSavedValue(key: string, defaultValue: any): any {
 const savedValue = localStorage.getItem(key);
 return JSON.parse(savedValue) || defaultValue;
}

function useLocalStorage(key: string, initialValue: any): [any, (value: any) => void] {
 const [value, setValue] = useState(getSavedValue(key, initialValue));
 useEffect(() => localStorage.setItem(key, JSON.stringify(value)), [key, value]);
 return [value, setValue];
}

export default useLocalStorage;
```

useContext

useContext

Хук позволяющий получать значения из компонента-провайдера на любом уровне вложенности.

useContext: createContext

- Как прокинуть данные в компоненты с глубокой вложенностью?
- Как изменить данные в родительском компоненте?
- Как повлиять на соседний компонент?

```
// импортируем метод createContext из React
import { createContext } from 'react';
// создаём и типизируем начальное состояние контекста
const initialContextValue: State = {
    state: [],
    title: ‘Title’
}
// наша переменная для формирования 'обёртки'
контекста
const stateContext =
createContext(initialContextValue);
// экспортируем контекст
export default stateContext;
```

useContext: Provider

```
// App.tsx
import stateContext from '../../context/stateContext';
import List from '../List/List';

export function App(): JSX.Element {
 return (
   // наш контекст становится компонентом-провайдером имеющий свойство
‘value‘, оно принимает значения, которыми будут пользоваться потребители
    <stateContext.Provider value={{ state: 'Example!' }}>
     <List />  // потребитель
    </stateContext.Provider>
 )
}
```

useContext

```
// List.tsx (один из потребителей)
import { useContext } from 'react';
import stateContext from '../../context/stateContext';

export function List(): JSX.Element {
 // используя деструктуризацию и хук useContext извлекаем ‘state’,
изначально он появился в компоненте stateContext.Provider внутри ‘value’
 const { state } = useContext(stateContext)

 return (
     <h1>{state}</h1>
 );
}
```

useReducer

useReducer

Хук для работы с централизованным состоянием приложения. Архитектурно схож с библиотекой Redux.

useReducer

```
const [state, dispatch] = useReducer(reducer, initialArg, init)
```

- `reducer` — чистая функция, которая принимает предыдущее состояние и действие (state и action), всегда возвращает новую версию состояния
- `initialArg` – аргумент для функции ‘init’, опциональный параметр.
- `init` – функция, которая принимает в качестве параметра ‘initialArg’, формируя тем самым исходное состояние приложения. Если параметра ‘initialArg’ нет, воспринимается как переменная.
- `state` – переменная которая хранит текущее состояние приложения, формируется на основе исходного состояния. Обычно в виде объекта.
- `dispatch` – функция взаимодействующая с reducer’ом, принимает в качестве параметра объект с действием (action) и полезной нагрузкой. Только dispatch может менять state.

useReducer: reducer

```ts
// types/User.ts
export default interface User {
 id: number;
 name: string;
}
export type UserId = User['id'];

// types/State.ts
export default interface State {
 users: User[];
 something: string;
}

// types/Action.ts
type Action =
 | { type: 'ADD_USER'; payload: User }
 | { type: 'REMOVE_USER'; payload: UserId };

export default Action;

// App.tsx
const [state, dispatch] = useReducer(reducer, {
 users: [],
 something: 'Elbrus',
});
```

```ts
// reducer.ts
export const reducer = (
 state: State,
 action: Action
): State => {
 switch (action.type) {
   case 'ADD_USER':
     return {
       ...state,
       users: [...state.users, action.payload],
     };
   case 'REMOVE_USER':
     return {
       ...state,
       users: state.users.filter(
         (el) => el.id !== action.payload
       ),
     };
 }

 return state;
};
```

useReducer: что нельзя?

- напрямую изменять то, что пришло в аргументах функции (state, action)
- выполнять какие-либо сайд-эффекты: например запросы к API
- вызывать не чистые функции: например Date.now() или Math.random()
- игнорировать копирование состояния

memo*

memo

Компонент высшего порядка (`higher order component`, `hoc`), обёртка для оптимизации рендеринга других компонентов. Работает по принципу мемоизации и только с примитивными типами.

Терминология: HOC

Higher order component

Компонент высшего порядка, HOC

Любая функция, принимающая компонент и возвращающая новый компонент.

`https://ru.reactjs.org/docs/higher-order-components.html`

Терминология: Memoization

Memoization, мемоизация

Подход к оптимизации программ. Сохранение результатов выполнения функции и их возврат в случае, если входные параметры не изменились.

memo

Предотвращает повторный рендеринг (выполнение функции), если props остались прежние.

Повышает производительность при динамическом изменении списка компонентов.

```
import { memo } from 'react';

function Some(
  { title }: { title: string }
): JSX.Element {
 return (
   <>
     <h1>{title}</h1>
   </>
 );
}

export default memo(Some);
```

memo: example code

```tsx
// App.tsx (родительский компонент для Some)
import { useState } from 'react';
import Some from '../Some/Some';

export function App(): JSX.Element {
 const [counter, setCounter] = useState(0);
 const title = 'Hello, React!';

 return (
   <>
     <button onClick={
    () => setCounter(counter + 1)}>
    Click: {counter}</button>
     <Some title={title} />
   </>
 );
}
```

```tsx
// Some.tsx (компонент с примитивным props)
import { memo } from 'react';

function Some(
  { title }: { title: string }
): JSX.Element {
 return (
   <>
     <h1>{title}</h1>
   </>
 );
}

export default memo(Some);
```

memo & useContext

Когда значение контекста меняется, все потребители рендерятся заново, независимо от `memo`.

useMemo*

useMemo

Хук принимающий “создающую” функцию и массив зависимостей. `useMemo` будет повторно вычислять мемоизированное значение только тогда, когда значение какой-либо из зависимостей изменилось. Эта оптимизация помогает избежать дорогостоящих вычислений при каждом рендере.

useMemo

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Помните, что функция, переданная в `useMemo`, запускается во время рендеринга. Не делайте в ней ничего, что вы обычно не делаете во время рендеринга. Например, побочные эффекты принадлежат только `useEffect`, а не `useMemo`.

`useMemo` всегда возвращает мемоизированное значение.

Если массив не был передан, новое значение будет вычисляться при каждом рендере.

useMemo, example code

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

export function MyHeading(): JSX.Element {
  const [count, setCount] = useState(0);
  const [cssStyle, setCssStyle] = useState(false);
  const heavyFunction = (number) => {
    console.log('Current value count: ', number);
    for (let index = 0; index < 99999999; index++) {}
    return number * number;
  }
  const countSquare = useMemo(() => heavyFunction(count), [count]);
  return (
    <>
      <h1 onClick={() => setCssStyle(!cssStyle)} style={{color: cssStyle ?
'green' : 'red'}}>useMemo</h1>
      <button onClick={() => setCount(count + 1)}>{count}</button>
      <h2>{countSquare}</h2>
    </>
  )
}
```

useCallback*

useCallback

Принимает встроенный callback и массив зависимостей. Хук `useCallback` вернёт мемоизированную версию callback’а, который изменяется только, если изменяются значения одной из зависимостей.

useCallback

```
const memoizedCallback = useCallback(() => doSomething(a, b), [a, b]);
```

Когда в массив зависимостей `useEffect` вы передаёте функцию, то при ре-рендере компонента ссылка на ячейку памяти этой функции меняется и для `useEffect` это новая функция (изменённая), значит `useEffect` сработает снова. Чтобы такого не происходило используют мемоизированный `callback`.

`useCallback` всегда возвращает мемоизированный `callback`.

useCallback, example code

```
import React, { useState, useEffect, useCallback } from 'react';

export function MyHeading(): JSX.Element {
  const [text, setText] = useState('');
  const [count, setCount] = useState(0);

  const addInConsole = useCallback((message) => console.log(message), []);

  useEffect(() => {
    addInConsole(text);
  }, [addInConsole, text]);

  return (
    <>
      <input type="text" onChange={(event) => setText(event.target.value)}
value={text}/>
      <button onClick={() => setCount(count + 1)}>Already {count} clicks</button>
    </>
  );
}
```

useCallback + useMemo

`useCallback` + `useMemo` позволяют организовать точечный рендеринг только тех компонентов, которые реально поменялись, даже если используется `useContext`.