Hooki

useState

```
const [title, setTitle] = useState('Tytuł domyślny');

return (
  <div>
    <h1>{title}</h1>
    <button onClick={() => setTitle('Nowy tytuł')} />Zmień tytuł</button>
  </div>
);
```

useEffect

```
useEffect(() => {
  ...                    // uruchamia się raz w momencie zamontowania komponentu
}, []);

return ....;
```

```
useEffect(() => {
  ...                    // uruchamia się podczas każdego renderowania
});

return ....;
```

```
useEffect(() => {
  ...                    // uruchamia się po zmianie wartości „title"
}, [title]);

return ....;
```

```
useEffect(() => {
  ...
  return () => {...}  // uruchamia się raz przed odmontowaniem komponentu
}, []);

return ....;
```

useReducer

```
const initialState = { title: 'Tytuł domyślny' };

function reducer(state, action) {
  switch (action.type) {
    case 'change-title': return { title: action.title }
    default: return state;
  }
}

const [state, dispatch] = useReducer(reducer, initialState);

return (
  <div>
    <h1>{state.title}</h1>
    <button
     onClick={() => dispatch({ type: 'change-title', title: 'Nowy tytuł'})} />
       Zmień tytuł
    </button>
  </div>
);
```

useContext

```
const ThemeContext = useContext({ color: '' });

.....

function App() {
  return (
    <ThemeContext.Provider value={{ color: 'red' }}>
      <Header>
    </ThemeContext.Provider>
  );
}

.....

function Header(props) {
  const theme = useContext(ThemeContext);

  return (
    <div style={{ color: theme.color }}>Witaj</div>
  );
}
```

useCallback

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

return (
  <div>
    <button onClick={() => memoizedCallback()}>Wykonaj</button>
  </div>
);
```

useMemo

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

return (
  <div>
    <h1>Skomplikowany wynik funkcji add: {memoizedValue}</h1>
  </div>
);
```

useRef

```
const refContainer = useRef(null);

const focusInput = () => refContainer.current.focus();

return (
  <div>
    <input ref={refContainer} />
  </div>
);
```

useLayoutEffect

```
useEffect(() => {
  ...                    // działa jak useEffect, ale wstrzymuje renderowanie widoku
}, []);

return ....;
```

useDebugValue

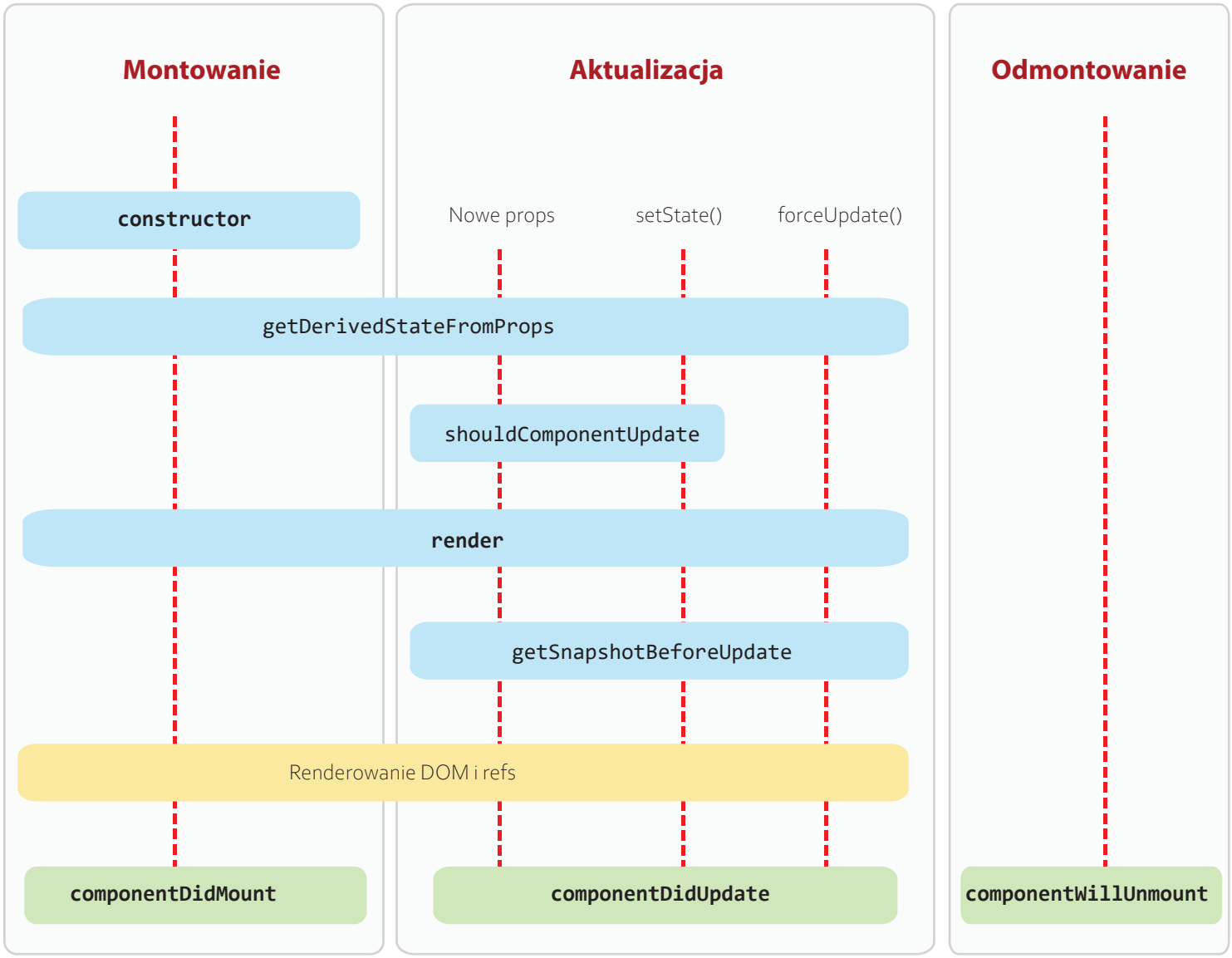
```
function useMyCustomAuthHook() {
  const [authenticated, setAuthenticated] = useState(false);

  useDebugValue(authenticated ? 'Zalogowany' : 'Wylogowany');
  ....
}                                    // tekst widoczny w React Developer Tools
```

useImperativeHandle

```
function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);
```

Cykle życia komponentu



zrodlo: https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

Komponenty funkcyjne vs klasowe

Przykłady

```
class Header {
  constructor(props) {
    this.state = { name: 'Jan' };
  }

  render () {
    return (
      <div>
        <h1>Witj {this.state.name} na stronie {this.props.website}</h1>;
        <button onClick={() => this.setState({name: 'Kamil'})}>
          Zmień imię
        </button>
      </div>
    );
  }
}
```

vs

```
function Header(props) {
  const [name, setName] = useState('Jan');

  return (
    <div>
      <h1>Witaj {name} na stronie {props.website}</h1>
      <button onClick={() => setName('Kamil')}>Zmień imię</button>
    </div>
  );
}
```

```
class Items {
  state = { items: [] };

  async fetchDataFromBackend() {
    const res = await axios.get('...');
    this.setState({ items: res.data });
  }

  componentDidMount() {
    this.fetchDataFromBackend();
  }

  render () {
    return (
      <div>
        {items.map(item => <div key={item.id}>{item.name}</div>)}
      </div>
    );
  }
}
```

vs

```
function Items(props) {
  const [items, setItems] = useState([]);
  const fetchItems = async () => {
    cont res = await axios.get('...');
    setItems(res.data);
  }

  useEffect(() => {
    fetchItems();
  }, [fetchItems]);

  return (
    <div>
      {items.map(item => <div key={item.id}>{item.name}</div>)}
    </div>
  );
}
```

```
class Items {
  cleanUp() {
    ....
  }

  componentWillUnmount() {
    this.cleanUp();
  }

  render () {
    return ...;
  }
}
```

vs

```
function Items(props) {
  const cleanUp = () => {...}

  useEffect(() => {
    return () => cleanUp();
  }, []);

  return ...;
}
```

Myślenie reactowe

Krok 1: Podziel interfejs użytkownika na zhiierarchizowany układ komponentów

tzn. podziel na komponenty i nadaj im nazwy

Krok 2: Zbuduj wersje statyczną

tzn. zakoduj same widoki

Nazwy komponentów reactowych często biorą się z nazw nadanych warstwom w Photoshopie
Zasada jednej odpowiedzialności, zgodnie z którą każdy komponent powinien być odpowiedzialny za tylko jedną rzecz
Nie należy używać stanu do budowy statycznych wersji aplikacji. Stan wiąże się wyłącznie z interaktywnością, tzn. danymi zmieniającymi się w czasie.
Zazwyczaj, budując proste aplikacje, zaczyna się od góry, natomiast w przypadku projektów większych łatwiej jest zacząć pracę od dołu hierarchii, jednocześnie pisząc testy dla poszczególnych funkcjonalności.

Krok 3: Określ minimalne (ale kompletne) odwzorowanie stanu interfejsu użytkownika

tzn. zaplanuj stan (*state*)

Krok 4: Określ, gdzie powinien mieścić się stan

tzn. dodaj stan (*state*) do wybranego komponentu

Stan zawsze idzie z góry na dół. Jak rzeka.

Krok 5: Dodaj przepływ danych w drugą stronę

tzn. dodaj akcje zmieniające stan (np. onClick, onChange...)