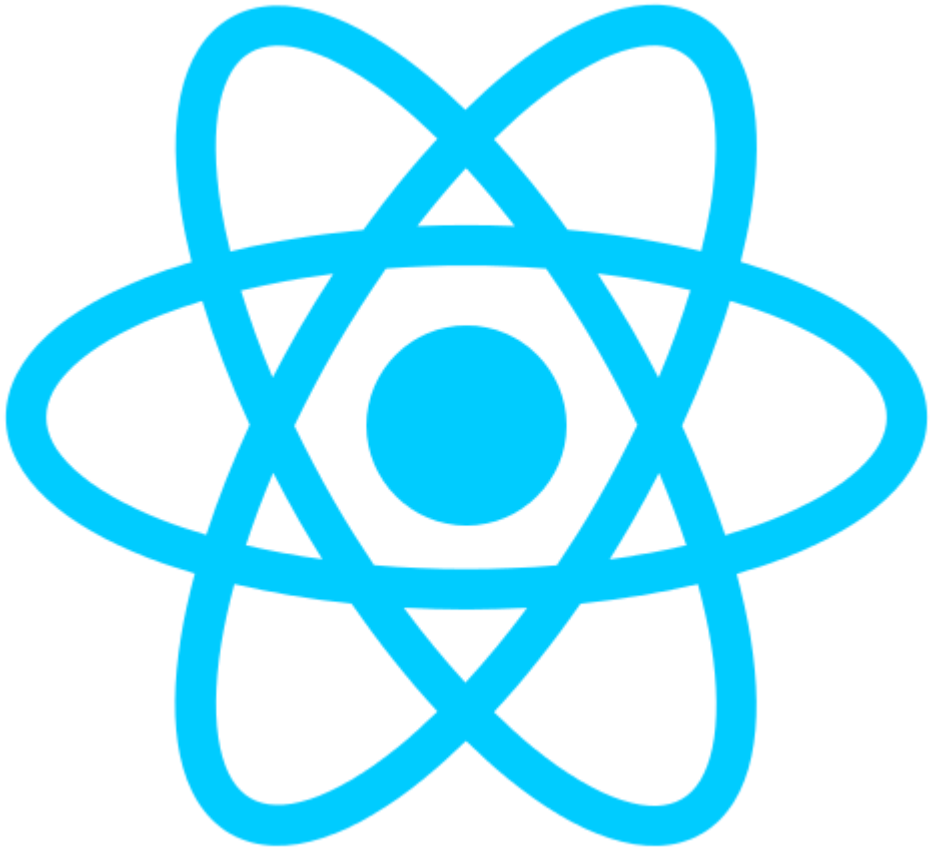
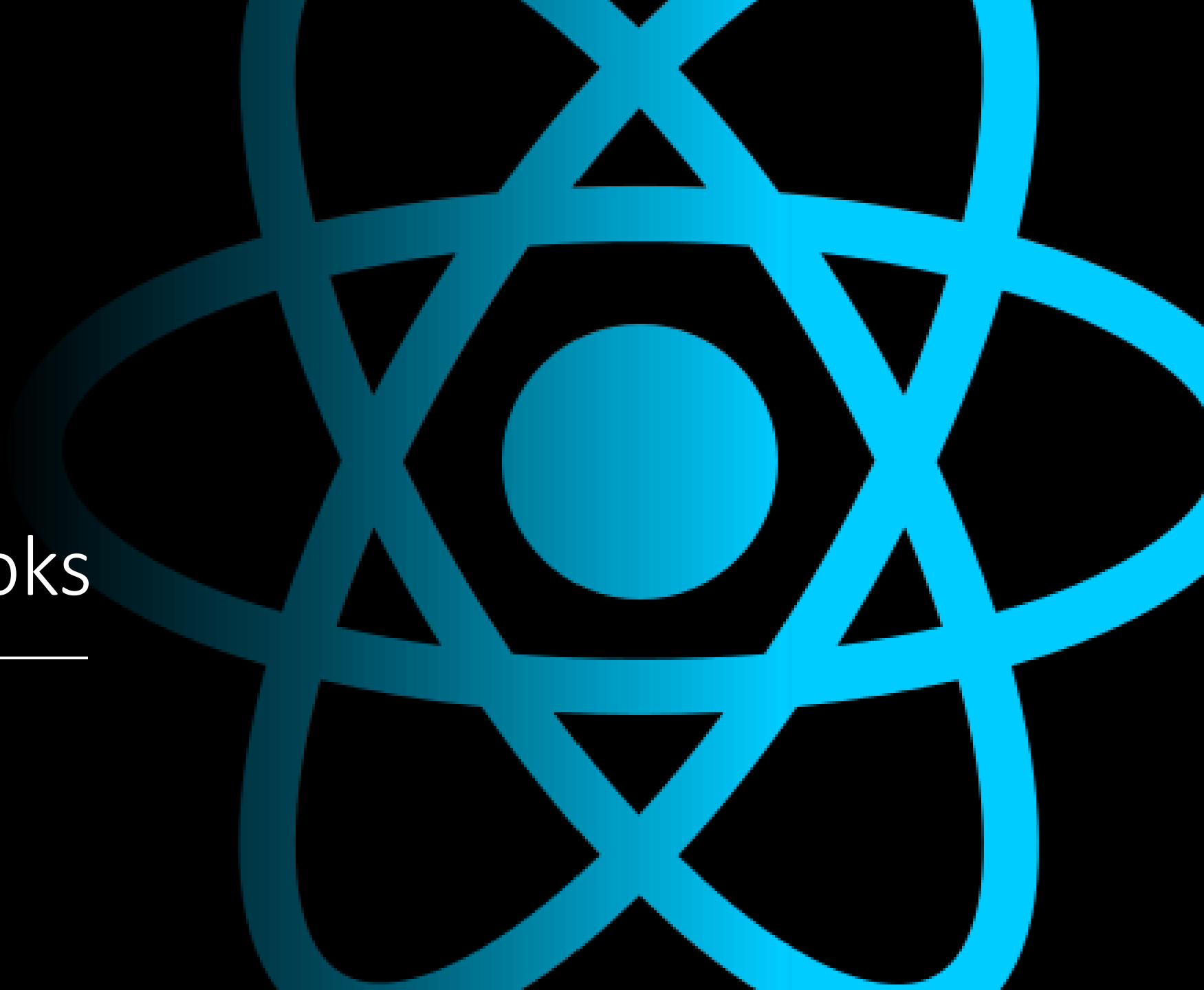


1. Wstęp do hooks
2. useState
3. useEffect

Wstęp do hooks



Czym są "hooki" w React?

Są to **Funkcje JavaScriptowe** pozwalające zaczepić w komponentach funkcyjnych rzeczy które do tej pory działały tylko w komponentach klasowych takie jak:

- stan komponentu
- cykl życia komponentu
- i wiele innych...

Okazuje się, że komponenty funkcyjne również mogą być "Stateful", tak więc mogą zachowywać się dokładnie tak samo jak komponenty klasowe.

To właśnie umożliwiają nam Hooki w React (od wersji 16.8).

Bardzo obszerna dokumentacja dotycząca Hooków znajduje się tutaj: Hooks Intro.

Podczas tej prezentacji skupimy się na dwóch najważniejszych Hookach w React:

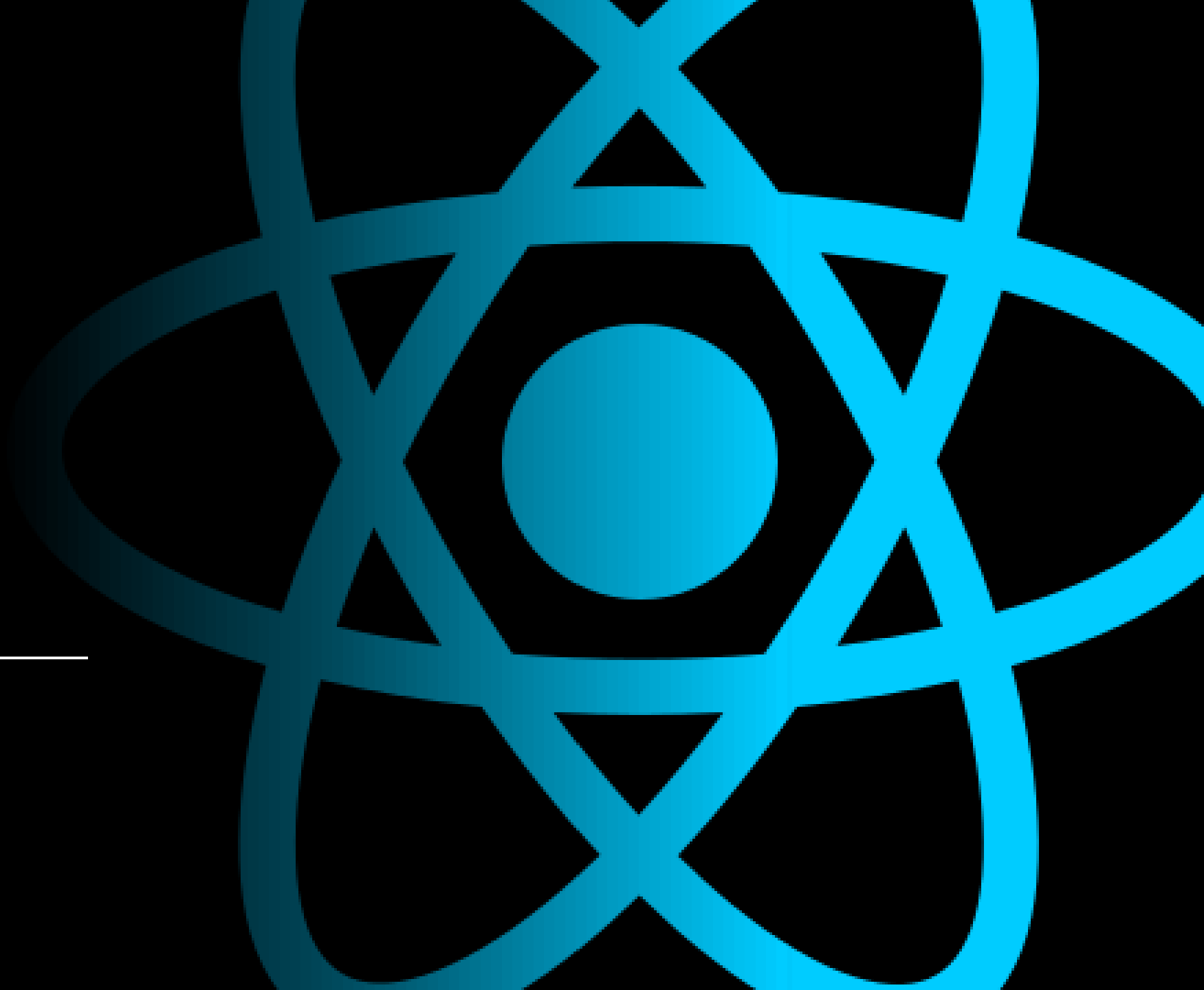
1. `useState`
2. `useEffect`

Zasady

Kiedy zaczniemy używać Hooków, musimy trzymać się kilku ważnych zasad:

1. Hook może być użyty **tylko w komponencie funkcyjnym lub w innym Hooku**.
   Nie możemy więc używać Hooków w funkcjach nie związanych z komponentem Reactowym.
2. Hooki **nie działają** w komponentach klasowych.
3. Nie definiuj Hooków w pętlach, warunkach czy zagnieżdżonych funkcjach.

useState



Jest to podstawowy Hook w React pozwalający nam zaczepić stan wewnętrzny do komponentu.

Aby skorzystać z metody `useState` należy zaimportować ją z biblioteki `"react"`:

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

Aby utworzyć stan początkowy trzeba uruchomić metodę `useState` i jako parametr przesłać wartość początkową. W naszym przypadku będzie to `0`.

```
const Counter = () => {
  const state = useState(0);
  return null;
}
```

Co pojawi się w zmiennej `state`?

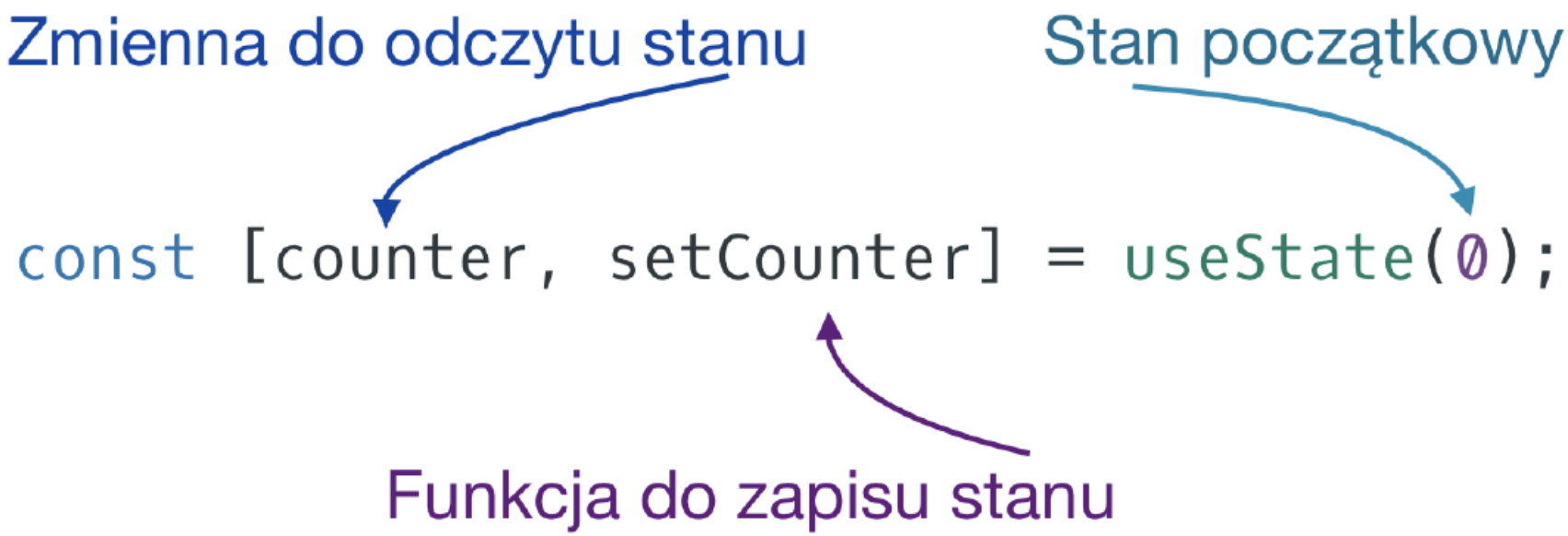
Będzie to tablica z dwoma elementami:

- aktualnym stanem (0)
- funkcją do zmiany stanu

*Należy pamiętać, że `useState` służy do stworzenia **pojedynczego** stanu komponentu. Można go przyrównać do **jednego** klucza w obiekcie `this.state` komponentu klasowego.*

Dzięki temu, że metoda `useState` zwraca tablicę elementów, skorzystamy z **destrukturyzacji** aby wyciągnąć te elementy do dwóch osobnych dobrze nazwanych zmiennych:

```
const Counter = () => {
  const [counter, setCounter] = useState(0);
  return null;
}
```



Dzięki temu, pod zmienną `counter` mamy aktualną wartość stanu komponentu, a pod zmienną `setCounter` funkcję dzięki której będziemy mogli aktualizować stan komponentu.

Nazwy zmiennych które tutaj wpisaliśmy są **dowolne**! Jednak powinny one dobrze opisywać dane które przetrzymują.

Pamiętaj, że `useState` zawsze zwraca elementy w takiej kolejności jak pokazano na przykładzie!

Jak teraz skorzystać z tego co zwróciła nam metoda `useState`?

Schemat działania jest praktycznie taki sam jak w przypadku komponentu klasowego.

Spróbujmy wyświetlić zmienną `counter`:

```
const Counter = () => {
  const [counter, setCounter] = useState(0);
  return (
    <h1>Liczba kliknięć: {counter}</h1>
  );
};
```

Kiedy wyrenderujemy komponent `Counter` na stronie pojawi się element `h1` z wiadomością: "Liczba kliknięć: 0".

A jak skorzystać z drugiej zmiennej w tym przykładzie czyli `setCounter`?

Tak jak mówiliśmy, jest to funkcja dzięki której możemy **zaktualizować** konkretny stan komponentu.

Aby to zrobić, wystarczy, że uruchomimy tę metodę w odpowiednim miejscu i jako parametr prześlemy do niej **nową wartość**.

Aktualizacja wartości stanu

```
setCounter(5)
```

Wartość w naszym stanie zmieni się z `0` na `5`.

A co za tym idzie, nasz komponent zostanie **wyrenderowany ponownie**.

Aktualizacja wartości stanu - ciąg dalszy

Jednak co w przypadku w którym **nowy stan** jest zależny od **poprzedniej wartości**?

W komponencie `Counter` właśnie zachodzi taka sytuacja. Do istniejącej już wartości chcemy dodać `1` i zapisać ją z powrotem.

W komponencie klasowym, używaliśmy metody `setState` do której przesyłaliśmy funkcję. Jako jej parametry otrzymywaliśmy między innymi **poprzedni stan (`prevState`)**.

W przypadku metody zwracanej przez `useState` jest bardzo podobnie!

W naszym przykładzie, użyjemy metody `setCounter` do której przekażemy funkcję. Poda nam ona jako parametr ostatni **aktualny** stan.

My z kolei zwrócimy z tej funkcji modyfikację tego stanu.

```
setCounter(prevState => prevState + 1);
```

Kiedy więc używać metody zwracanej z `useState` bez a kiedy z dodatkową funkcją?

Jest na to prosta zasada:

Jeżeli metoda modyfikująca stan nie jest od niego zależna (np. jednorazowa zmiana wartości `true` na `false`) to nie musimy używać funkcji wyższego rzędu.

W każdym innym przypadku powinniśmy z niej skorzystać i odczytywać **aktualną wartość stanu** właśnie z tej funkcji.

Dlaczego jest to ważne? Ze względu na to, że metody modyfikujące stan działają **asynchronicznie**! Dokładnie tak jak metoda `setState`. Dlatego właśnie musimy mieć zawsze pewność, że modyfikujemy poprawną wartość stanu.

Użycie bez funkcji

```
const TurnOff = () => {
  const [state, setState] = useState(true);
  const handleClick = () => {
    setState(false);
  }
  return (
    <button onClick={handleClick}>
      Turn Off!
    </button>
  );
};
```

Użycie z funkcją

```
const Counter = () => {
  const [counter, setCounter] = useState(true);
  const handleClick = () => {
    setCounter(prevState => prevState + 1);
  }
  return (
    <>
      <h1>Liczba kliknięć: {counter}</h1>
      <button onClick={handleClick}>
        Kliknij!
      </button>
    </>
  );
};
```

Metoda zmieniająca stan korzysta z `prevState` który jest **aktualną** reprezentacją wartości zmiennej `counter`.

Zanim wrócimy do naszego przykładu, musimy wspomnieć o jednej różnicy między metodą zwracaną przez `useState` a metodą `setState` z komponentów klasowych.

Metoda `setState` potrafiła automatycznie **połączyć** poprzednie klucze i wartości w stanie komponentu i nadpisać tylko te nowe.

W przypadku Hooka `useState` musimy zadbać o to sami. Metoda zmieniająca stan zawsze po prostu podmienia stan. Jest to szczególnie istotne kiedy przetrzymujemy w stanie obiekt.

```
const User = () => {
  const [user, setUser] = useState({
    name: "John",
    surname: "Doe",
    age: 20
  });

  handleChange = () => {
    setUser(prevState => {
      return {
        ...prevState,
        name: "Mark"
      }
    });
  };

  // reszta logiki komponentu
};
```

Musimy rozproszyć obiekt ze stanem a następnie nadpisać wartości które chcemy zmodyfikować.

Wróćmy do naszego przykładu

Brakuje w nim miejsca w którym moglibyśmy uruchomić metodę zmieniającą stan. Dodajmy tutaj `button` i metodę `handleClick`.

```
const Counter = () => {
  const [counter, setCounter] = useState(0);

  const handleClick = () => {
    setCounter(prevState => prevState + 1);
  };

  return (
    <>
      <h1>Liczba kliknięć: {counter}</h1>
      <button onClick={handleClick}>Kliknij!</button>
    </>
  );
};
```

Metoda którą zwróciło nam `useState` ma w sobie odpowiednie mechanizmy dzięki którym następuje ponowne renderowanie komponentu.

Zmieniać stan komponentu możemy tylko i wyłącznie poprzez dedykowaną do tego metodę!

Zasada ta jest taka sama jak w przypadku komponentów klasowych. Tam musieliśmy zawsze używać `setState` do aktualizacji stanu. Tutaj musimy używać metod które zwraca `useState`.

Przeanalizujmy co się wydarzyło w tym przykładzie.

1. W komponencie `Counter` utworzyliśmy stan za pomocą metody `useState`
2. Wyciągnęliśmy dwie wartości: `counter` (wartość stanu), `setCounter` (funkcję zmieniającą stan)
3. Stworzyliśmy metodę pomocniczą `handleClick` która uruchamia metodę `setCounter` przesyłając jako parametr **aktualną wartość** stanu zwiększoną o `1`
4. Następnie zwróciliśmy z komponentu konstrukcję zawierającą wyświetlenie licznika a także przycisk uruchamiający metodę `handleClick` po kliknięciu a co za tym idzie zwiększanie licznika.

Oto porównanie osiągnięcia tej samej funkcjonalności przez komponent funkcyjny i komponent klasowy.

Funkcyjnie z React Hooks

```
const Counter = () => {
  const [counter, setCounter] = useState(0);

  const handleClick = () => {
    setCounter(prevState => prevState + 1);
  };

  return (
    <>
      <h1>Liczba kliknięć: {counter}</h1>
      <button onClick={handleClick}>
        Kliknij!
      </button>
    </>
  );
};
```

Klasowo z setState

```
class Counter extends Component {
  state = {
    counter: 0
  }

  handleClick = () => {
    this.setState((prevState) => ({
      counter: prevState.counter + 1
    }));
  }

  render() {
    return (
      <>
        <h1>Liczba kliknięć: {this.state.counter}</h1>
        <button onClick={this.handleClick}>
          Kliknij!
        </button>
      </>
    );
  }
}
```

Powiedzieliśmy wcześniej, że metoda `useState` zaczepia pojedynczy stan dla komponentu. A co w sytuacji w której chcielibyśmy przetrzymywać więcej danych które mogły by się zmieniać?

Możemy tę metodą uruchamiać w komponencie wielokrotnie. Oto przykład:

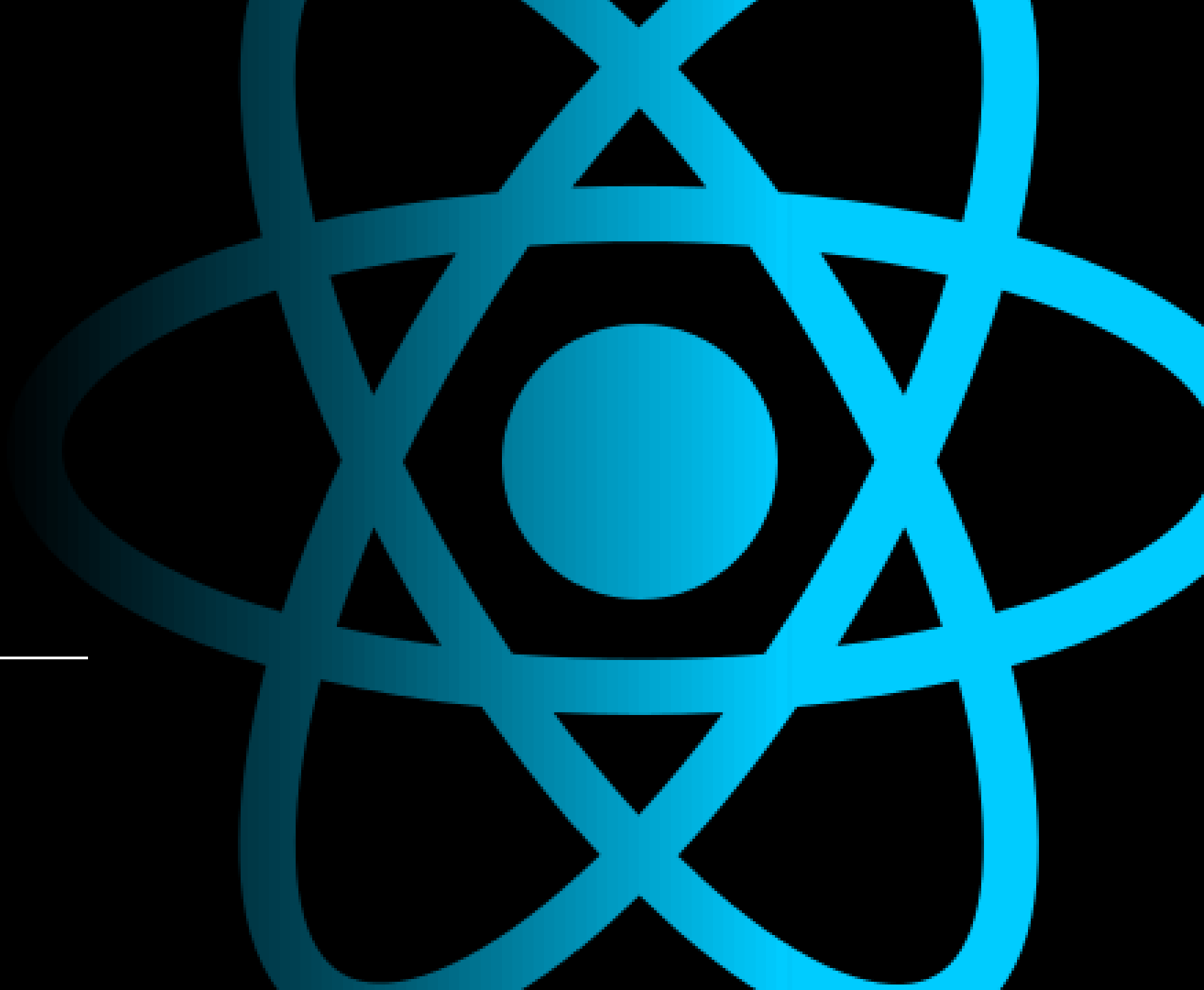
```
const Cart = () => {
  const [sum, setSum] = useState(0);
  const [products, setProducts] = useState([]);
  const [invoice, setInvoice] = useState(false);

  // reszta logiki aplikacji
}
```

Stworzyliśmy tutaj trzech osobne stany dla trzech różnych rzeczy:

1. `sum` - suma do zapłacenia za towary
2. `products` - lista produktów w formie tablicy
3. `invoice` - informacja `true/false` o tym czy klient potrzebuje fakturę do zamówienia

Każdy z tych stanów ma **dedykowaną** metodę do zarządzania nim.

useEffect



Jest to kolejny z podstawowych Hooków w React. Jego podstawowym funkcją jest wykonywanie efektów ubocznych w komponentach funkcyjnych.

Możemy do nich zaliczyć:

- setTimer, setInterval
- Pobieranie danych z serwerów zewnętrznych
- Podpinanie się pod wydarzenia (np. `resize`)
- Aktualizacja obiektu `document`

Możemy przyjąć (upraszczając), że metoda `useEffect` jest tym dla komponentu funkcyjnego co `componentDidMount`, `componentDidUpdate` i `componentWillMount` dla komponentu klasowego.

Będziemy mogli przeprowadzić w nim wszystkie operacje w prostszy i mniej powtarzalny sposób niż to było w przypadku komponentów klasowych i ich metod cyklu życia!

Z czego się składa?

Metoda `useEffect` również powinna być zaimportowana z biblioteki `react`.

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

Następnie aby jej użyć, potrzebujemy w większości przypadków stan komponentu na który będziemy reagować.

Tak więc używamy tej metody **już po inicjalizacji stanu komponentu**!

Jako jej pierwszy parametr przesyłamy **funkcję**, która zostanie uruchomiona **po** wyrenderowaniu komponentu. Dokładnie tak jak `componentDidMount`.

Przykład

```
const App = () => {
  const [counter, setCounter] = useState(1);

  useEffect(() => {
    console.log("Komponent zamontowany!");
  });

  console.log("Render Komponentu!");
  return <h1>{counter}</h1>
};
```

A co się wydarzy jeżeli nasz komponent zacznie **zmieniać swój stan**?

Dodajmy tutaj przycisk zwiększający licznik `counter`.

Okazuje się, że metoda `useEffect` w takiej formie, zadziała również tak jak `componentDidUpdate` czyli będzie uruchomiona **przy każdej aktualizacji** komponentu.

W konsoli pojawią nam się obie wiadomości w momencie renderowania, po zamontowaniu i każdorazowej aktualizacji.

```
const App = () => {
  const [counter, setCounter] = useState(1);
  useEffect(() => {
    console.log("Komponent zamontowany/zaktualizowany");
  });

  const handleClick = () => {
    setCounter(prevState => prevState + 1);
  };

  console.log("Render Komponentu");
  return (
    <>
      <h1>{counter}</h1>
      <button onClick={handleClick}>
        Klik
      </button>
    </>
  );
};
```

Widzimy więc, że `useEffect` w swojej najbardziej podstawowej konstrukcji działa podobnie jak:

- `componentDidMount`
- `componentDidUpdate`

Uruchamia się przy zamontowaniu a także przy każdej aktualizacji: kiedy zmieni się stan komponentu, lub kiedy dostanie nowy zestaw propsów.

Jednak mówiliśmy, że `useEffect` jest przydatny szczególnie w efektach ubocznych. Takim efektem ubocznym jest np. interwał.

Spróbujmy w `useEffect` wstawić interwał który co sekundę zwiększy licznik o 1.

Nie uruchamiajcie tego kodu! Dlaczego? Program wpadnie w **nieskończoną pętlę**.

Przed chwilą powiedzieliśmy, że `useEffect` będzie uruchamiany i przy montowaniu komponentu a także przy **jego aktualizacji**!

Prześledźmy co się dzieje:

```
const App = () => {
  const [counter, setCounter] = useState(1);

  useEffect(() => {
    const interval = setInterval(() => {
      setCounter(prevState => prevState + 1);
    }, 1000)
  });

  return <h1>{counter}</h1>;
};
```

Nie uruchamiajcie tego kodu! Dlaczego? Program wpadnie w **nieskończoną pętlę**.

Przed chwilą powiedzieliśmy, że `useEffect` będzie uruchamiany i przy montowaniu komponentu a także przy **jego aktualizacji**!

Prześledźmy co się dzieje:

1. Zostaje ustawiony stan na `1`
2. Renderujemy widok komponentu
3. Uruchamia się `useEffect`
4. Włączany jest interwał (działający co 1 sekundę)
5. Zwiększamy licznik o 1
6. I cały program zaczyna się od nowa!

```
const App = () => {
  const [counter, setCounter] = useState(1);

  useEffect(() => {
    const interval = setInterval(() => {
      setCounter(prevState => prevState + 1);
    }, 1000)
  });

  return <h1>{counter}</h1>;
};
```

Po aktualizacji komponentu, `setInterval` uruchamiany jest ponownie. Po chwili liczba interwałów rośnie do takiego rozmiaru, że karta przeglądarki zostaje zawieszona.

Jak temu zapobiec? W jaki sposób "zmusić" `useEffect` aby uruchamiał się tylko przy **zamontowaniu komponentu**?

`useEffect` jako drugi parametr przyjmuje tablicę elementów. W tablicy tej możemy umieścić zmienne, które - jeżeli ulegną zmianie - spowodują ponowne uruchomienie przesłanej do `useEffect` funkcji.

W naszym przykładzie chcemy żeby uruchomiła się ona tylko raz, dlatego **jako drugi parametr wstawimy pustą tablicę**!

```
const App = () => {
  const [counter, setCounter] = useState(1);

  useEffect(() => {
    const interval = setInterval(() => {
      setCounter(prevState => prevState + 1);
    }, 1000)
  }, []);

  return <h1>{counter}</h1>;
};
```

Tutaj pojawił się **drugi parametr** metody `useEffect` - pusta tablica.

Powiedzieliśmy przed chwilą, że tablica którą przesyłamy do `useEffect` może być wypełniona zmiennymi. Po co?

Dzięki temu, możemy zdecydować co **dokładnie** wyzwoli naszą funkcję po **aktualizacji** komponentu.

Spójrzmy na przykład obok:

- Mamy tutaj **dwa stany** które możemy aktualizować: `counter` i `age`.
- W metodzie `useEffect` wykorzystujemy jednak tylko `counter` do tego aby zmienić tytuł naszej strony w obiekcie `document`.

Jak ograniczyć działanie `useEffect` tylko kiedy zostanie zaktualizowany `counter`?

```
const App = () => {
  const [counter, setCounter] = useState(1);
  const [age, setAge] = useState(25);

  useEffect(() => {
    console.log(counter);
    document.title = `Clicked: ${counter}`;
  });

  const handleCounter = () => {
    setCounter(prevState => prevState + 1);
  };

  const handleGetOlder = () => {
    setAge(prevState => prevState + 1);
  };

  return (
    <>
      <h1>Clicked: {counter}</h1>
      <h2>Age: {age}</h2>
      <button onClick={handleCounter}>Counter!</button>
      <button onClick={handleGetOlder}>Get Older!</button>
    </>
  );
};
```

Metoda `useEffect` musi się dowiedzieć, która ze zmiennych (stanu lub propsów) musi zostać zaktualizowana, aby uruchomiła metodę do niej przesłaną.

Zapisujemy to w tej samej tablicy, którą w poprzednim przykładzie zostawiliśmy pustą.

Wpisujemy tutaj zmienną `counter`, bo to **jej zmiana** ma **wyzwalać** `useEffect`!

Od teraz zmiana wartości w stanie `age` **nie będzie** uruchamiała ponownie `useEffect`.

```
const App = () => {
  const [counter, setCounter] = useState(1);
  const [age, setAge] = useState(25);

  useEffect(() => {
    console.log(counter);
    document.title = `Clicked: ${counter}`;
  }, [counter]);

  const handleCounter = () => {
    setCounter(prevState => prevState + 1);
  };

  const handleGetOlder = () => {
    setAge(prevState => prevState + 1);
  };

  return (
    <>
      <h1>Clicked: {counter}</h1>
      <h2>Age: {age}</h2>
      <button onClick={handleCounter}>Counter!</button>
      <button onClick={handleGetOlder}>Get Older!</button>
    </>
  );
};
```

Jest to ważne z poziomu optymalizacji. Nie chcemy aby wykonywały się funkcje w losowych i nie zamierzonych momentach. Jeżeli zmiana stanu miałaby wykonywać bardziej złożone operacje jak np. pobranie danych z serwera zewnętrznego, czy wykonanie skomplikowanych obliczeń to musimy być pewni, że będą się one wykonywać tylko wtedy kiedy jest to niezbędne.

A co jeżeli chcielibyśmy, żeby w naszym przykładzie `useEffect` reagował na zmianę wartości w `age`, ale w zupełnie inny sposób niż robi to `useEffect` dla `counter`?

Dodajemy następny `useEffect` w którym jako drugi parametr deklarujemy tablicę ze zmienną `age`. Bo to pod jej wpływem ma być uruchamiana metoda przesłana do tego hooka.

```
const App = () => {
  // state i effect dla counter
  const [counter, setCounter] = useState(1);
  useEffect(() => {
    console.log(counter);
    document.title = `Clicked: ${counter}`;
  }, [counter]);

  // state i effect dla age
  const [age, setAge] = useState(25);
  useEffect(() => {
    // akcje dla zmiany age
  }, [age]);

  // reszta logiki komponentu
};
```

Wiemy już, że metoda `useEffect` pozwala na reagowanie w momencie: zamontowania czy aktualizacji komponentu.

Jednak kiedy wrócimy do naszego przykładu z interwałem, okazuje się, że zapomnieliśmy o ważnej rzeczy. **Zawsze musimy po sobie sprzątać!**

W komponencie klasowym używaliśmy do tego metody `componentWillUnmount`. Jak będzie w przypadku `useEffect`?

Aby zareagować na od montowanie komponentu przy użyciu `useEffect`, wystarczy, że **zwrócimy w nim funkcję** która, np. wyczyści interwał czy timeout.

Za wywołaniem metody `setCounter` pojawiło się słowo `return` a następnie deklaracja funkcji (może to być jakakolwiek funkcja, niekoniecznie strzałkowa).

W funkcji ten uruchomiliśmy metodę `clearInterval` przekazując do niej wartość zmiennej `interval` która kryje w sobie ID naszego interwału.

Zwrócenie funkcji czyszczącej jest opcjonalne i zależy od tego, czy rzeczywiście w naszym komponencie musimy zadbać o wykasowanie interwałów, czy np. jakichś subskrypcji API.

```
const App = () => {
  const [counter, setCounter] = useState(1);

  useEffect(() => {
    const interval = setInterval(() => {
      setCounter(prevState => prevState + 1);
    }, 1000);

    return () => {
      clearInterval(interval);
    }
  }, []);

  return <h1>{counter}</h1>;
};
```

```jsx
const App = () => {
  const [counter, setCounter] = useState(1);

  useEffect(() => {
    const interval = setInterval(() => {
      setCounter(prevState => prevState + 1);
    }, 1000);
  }, []);

  return <h1>{counter}</h1>;
};
```

```jsx
class App extends Component {
  state = {
    counter: 1
  }

  componentDidMount() {
    this.interval = setInterval(() => {
      setState(prevState => {
        return {
          counter: prevState.counter + 1
        }
      }
    }, 1000);
  }

  render() {
    return <h1>{this.state.counter}</h1>;
  }
};
```

Uruchomienie interwału w odpowiedniej metodzie.

```jsx
const App = () => {
  const [counter, setCounter] = useState(1);

  useEffect(() => {
    const interval = setInterval(() => {
      setCounter(prevState => prevState + 1);
    }, 1000);
  }, []);

  return <h1>{counter}</h1>;
};
```

```jsx
class App extends Component {
  state = {
    counter: 1
  }

  componentDidMount() {
    this.interval = setInterval(() => {
      setState(prevState => {
        return {
          counter: prevState.counter + 1
        }
      }
    }, 1000);
  }

  render() {
    return <h1>{this.state.counter}</h1>;
  }
};
```

Przekazanie **pustej tablicy** do `useEffect` sprawi, że będzie on się zachowywał jak metoda `componentDidMount`.

```jsx
const App = () => {
  const [counter, setCounter] = useState(1);

  useEffect(() => {
    const interval = setInterval(() => {
      setCounter(prevState => prevState + 1);
    }, 1000);

    return () => {
      clearInterval(interval);
    }
  }, []);

  return <h1>{counter}</h1>;
};
```

```jsx
class App extends Component {
  state = {
    counter: 1
  }

  componentDidMount() {
    this.interval = setInterval(() => {
      setState(prevState => {
        return {
          counter: prevState.counter + 1
        }
      }
    }, 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return <h1>{this.state.counter}</h1>;
  }
};
```

Zwrócenie w funkcji `useEffect` metody czyszczącej jest tym co dla komponentu klasowego `componentWillUnmount`.

```jsx
const App = () => {
  const [counter, setCounter] = useState(1);

  useEffect(() => {
    document.title = `Clicked: ${counter}`;
  }, [counter]);

  const handleCounter = () => {
    setCounter(prevState => prevState + 1);
  };

  return (
    <>
      <h1>Clicked: {counter}</h1>
      <button onClick={handleCounter}>Counter!</button>
    </>
  );
};
```

Metoda `useEffect` pozwala na zapisanie powtarzającej się logiki **w jednym miejscu**. W przypadku komponentu klasowego musielibyśmy powtarzać dokładnie to samo w dwóch miejscach.

```jsx
class App extends Component {
  state = {
    counter: 1
  }

  componentDidMount() {
    document.title = `Clicked: ${this.state.counter}`;
  }

  componentDidUpdate() {
    document.title = `Clicked: ${this.state.counter}`;
  }

  handleCounter = () => {
    this.setState(prevState => ({
      counter: prevState.counter + 1
    }));
  };

  render() {
    return (
      <>
        <h1>Clicked: {this.state.counter}</h1>
        <button onClick={this.handleCounter}>Counter!</button>
      </>
    );
  }
};
```