



# REACT

## Zaawansowane zagadnienia

# Destrukturyzacja

ES6 często wykorzystywany w kodzie aplikacji React

# destrukuryzacja (destructuring)

Destrukuryzacja dotyczy obiektów i tablic

```
const player = {  
  age: 24,  
  name: "John"  
};
```

```
const players = ["Jacek", "Tomasz"];
```

# destrukturyzacja (destructuring)

Destrukturyzacja (od ES6) pozwala **wyodrębnić** informacje z tablic i obiektów. Destrukturyzacja jest procesem podziału struktury obiektu/tablicy na mniejsze fragmenty.

# destrukuryzacja obiektu

```
const player = {  
  age: 24,  
  name: "John"  
};
```

```
//ES5 - wyodrębnienie tradycyjne  
//const age = player.age;  
//const name = player.name;  
const age = player.age, name = player.name;
```

```
//ES6 - destrukuryzacja  
const {age, name} = player;  
//lub oczywiście z let jeśli jest taka potrzeba  
let {age, name} = player;
```

# destruktywizacja obiektu

```
const player = {  
  age: 24,  
  name: "John"  
};
```

```
//Możemy też spotkać taki zapis (dla niektórych czytelniejszy)  
const {  
  age,  
  name  
} = player;
```

# destruktywizacja tablicy

```
const players = ["Jacek", "Tomasz"];
```

//Podajemy nazwę do której będzie przypisany dana wartość  
tablicy (zgodnie z kolejnością)

```
const [user1, user2] = players;
```

```
//user1 = "Jacek"
```

```
//user2 = "Tomasz"
```

# Wykorzystanie w React

// Oszczędzamy czas, kod może być bardziej przejrzysty.

```
const {value, id, date, item, active} = this.state;
```

//alternatywne wyodrębnienie bez destrukuryzacji

```
const item = this.state.item;
```



# Destrukturyzacja obiektu - wartości domyślne

```
//Domyślne przypisanie wartości  
const { value = "", id, date, item, active = true } = user;
```

Przypisanie domyślne wartości, gdy:

- właściwość nie istnieje
- właściwość istnieje, ale ma wartość undefined

Jeśli nie ma wartości domyślnej, a nazwana właściwość nie istnieje, to powstaje zmienna z przypisaną wartością undefined.

# Destrukturyzacja obiektu - zmiana nazwy

```
const player = {  
  age: 24,  
  name: "John"  
}
```

```
const { age : playerAge, name : playerName} = player;
```

```
//Teraz wartości właściwości znajdują się w zmiennych  
playerAge i playerName.
```

# Destrukturyzacja tablicy - wybrane elementy

```
const players = ["Jacek", "Tomasz", "Jarek"]
```

//możemy tylko konkretny element wyodrębnić. Po co tworzyć zmienne, których nie potrzebujemy.

```
let [user1,,user3] = players;
```

```
let [, ,player3] = players;
```

# Element rest (jak parametr rest) ...

```
const players = ["Jacek", "Tomasz", "Jarek", "Arek"];
```

```
let [user1, ...users] = players;
```

```
user1 -> "Jacek"
```

```
users(3) -> ["Tomasz", "Jarek", "Arek"]
```

Element rest musi być na końcu przy destrukuryzacji tablicy.

Ps. jest to też kolejny sposób na zrobienie kopii tablicy

```
const [...newPlayers] = players;
```

```
newPlayers(4) -> ["Jacek", "Tomasz", "Jarek", "Arek"]
```

# Praca z formularzami

# Praca z formularzami

Technika kontrolowanego i niekontrolowanego komponentu do pracy z formularzami.

Controlled Component | Uncontrolled Component

**Technika**, ponieważ nie chodzi tu o specjalne właściwości jakiegoś nowego rodzaju komponentu, a pracę z formularzem (elementami formularza) w oparciu o **określone reguły**.

\* Kontrolowany/Niekontrolowany czyli czy komponent React ma odpowiadać za kontrolowanie zawartości stanu formularza.

# Kontrolowany komponent (formularz)

1. Wykorzystanie `state` do trzymania aktualnej wartości elementów formularza (`jedyne źródło prawdy`)
2. Wykorzystanie atrybutów `value/checked` jako określenie zawartości elementów formularza.
3. Zmiana właściwości `state` (czyli właściwości posiadającej aktualną wartości pola) za pomocą metody `onChange`

# Kontrolowany komponent (formularz)

```
state = {  
  city: ""  
}
```

```
<input  
  type="text"  
  value={this.state.city}  
  onChange={this.handleChange}  
>
```



# Jedno źródło prawdy (single source of truth)

HTML ma atrybut value, ale ma też taki atrybut react. Jeśli decydujemy się na kontrolowany komponent, to musimy użyć wartości value w polu.

```
<textarea  
  value={this.state.text}  
  onChange={this.handleChange}  
  name="text"  
>
```

# Wartość domyślna (początkowa) pola

```
state = {  
  text: "" //wartość inicjalizująca właściwości jest wartością początkową  
}
```

```
<input  
  type="text"  
  value={this.state.text}  
  onChange={this.handleChange}  
>
```

# Zmiana stanu z każdą zmianą w polu

```
state = {  
  text: ""  
}  
  
// obsługa zmiany właściwości w state (onChange na polu, onSubmit na formularzu)  
handleChange = e => {  
  this.setState({  
    text: e.target.value  
  })  
}  
  
* Dla <input type="checkbox"> czy  
  <input type="radio"> będzie to atrybut checked (a nie  
  value) i nie e.target.value a e.target.checked  
  
<input  
  type="text"  
  value={this.state.text}  
  onChange={this.handleChange}  
>
```

# Niekontrolowany komponent

Istnieje prostszy sposób na dostęp do elementu DOM. Jest nim użycie atrybutu ref (references) w polu formularza.

# Niekontrolowany komponent - atrybut reference

```
render() {  
  return (  
    <div>  
      <form onSubmit={this.handleSubmit}>  
        <input  
          type="text"  
          placeholder="wpisz imię"  
          ref="username"  
        />  
        <button type="submit">Potwierdź</button>  
      </form>  
    </div>  
  );  
}
```

# Niekontrolowany komponent - odwołanie (this.refs)

```
render() {  
  return (  
    <div>  
      <form onSubmit={this.handleSubmit}>  
        <input  
          type="text"  
          placeholder="wpisz imię"  
          ref="username"  
        />  
        <button type="submit">Potwierdź</button>  
      </form>  
    </div>  
  );  
}
```

```
handleSubmit = (e) => {  
  e.preventDefault()  
  const name = this.refs.username.value;  
  //dodanie gdzieś np. w state (jakaś tablica)  
  this.refs.username.value = ''; //wyzerowanie  
}
```

# Niekontrolowany komponent - domyślna wartość

```
render() {  
  return (  
    <div>  
      <form onSubmit={this.handleSubmit}>  
        <input  
          type="text"  
          placeholder="wpisz imię"  
          ref="username"  
          defaultValue="imię"  
        />  
        <button type="submit">Potwierdź</button>  
      </form>  
    </div>  
  );  
}
```

```
handleSubmit = (e) => {  
  e.preventDefault()  
  const name = this.refs.username.value;  
  //dodanie gdzieś np. tablicy w state  
  this.refs.username.value = '';  
}
```

imię	Potwierdź
------	-----------

\*Dla `<input type="checkbox">` czy  
`<input type="radio">` będzie to `defaultChecked`  
(true/false)

# Dostęp do właściwości przez event (alternatywa)

```
render() {  
  return (  
    <div>  
      <form onSubmit={this.handleSubmit}>  
        <input  
          type="text"  
          placeholder="wpisz imię"  
          name="username"  
        />  
        <button type="submit">Potwierdź</button>  
      </form>  
    </div>  
  );  
}
```

```
handleSubmit = (e) => {  
  e.preventDefault()  
  const name = e.target.elements.username.value;  
  //dodanie gdzieś np. tablicy w state  
  e.target.elements.username.value = '';  
}
```



# Kontrolowany vs. Niekontrolowany

- Kontrolowany pozwala na więcej (renderowanie, aktualizacje w innych miejscach w interfejsie) .
- Kontrolowany jest bardziej "reactowy" czyli praca ze stanem i nie manipulowanie DOM bezpośrednio - zostawiamy to React, w końcu do tego jest.

Kontrolowany komponent znacznie ułatwia walidację (o czym w przyszłości)

# Kontrolowany vs. Niekontrolowany

Niekontrolowany, to trochę mniej kody i szybszy dostęp do wartości pól/formularza.

Dokumentacja rekomenduje, a większość developerów React korzysta, z **techniki kontrolowanego komponentu**, źródłem prawdy dla komponentu jest wtedy obiekt state, a nie wartość DOM.

Zapamiętajmy: Źródłem prawdy dla komponentu jest state (kontrolowany) lub wartość DOM (niekontrolowany)

# setState

asynchroniczność i scalanie

# metoda setState - co wiemy

Stan ustawiamy za pomocą właściwości state (this.state w metodzie constructor) podczas inicjalizacji. Potem wszelkich zmian w state dokonujemy już za pomocą metody setState.

Przy wywołaniu metody setState domyślnie następuje ponowne renderowanie komponentu (i komponentów dzieci).

# metoda setState - asynchroniczność i scalanie

Zmiany w stanie (state) za pomocą metody setState.

Ponowne renderowanie komponentu (i jego komponentów dzieci)

Ale to nie wszystko co musimy wiedzieć...

- Aktualizacje stanu są **asynchroniczne** (metoda setState jest wywoływana asynchronicznie)
- React może **łączyć** (scalać) **wiele wywołań setState** w jedno.

# Pamiętajmy, że JavaScript jest jednowątkowy

Musimy w tym momencie przypomnieć sobie kilka cech JS, m.in. na czym polega **jednowątkowość**, czym jest **wywołanie zwrotne** i jak działa **pętla zdarzeń**.

# Pamiętajmy, że JavaScript jest jednowątkowy

JavaScript jest językiem **jednowątkowym** tzn. robi jedno zadanie w danym czasie, a dopiero potem (później) przechodzi do kolejnego zadania.

Wyobraźmy sobie jednak oczekiwanie na informacje zwrotną (np. pobranie danych z API), które trwa np. kilka sekund. Wyobraźmy sobie, że w tym czasie nasz program jest zawieszony (blokowany) tzn. nie przejdzie dalej dopóki nie skończy działania. Jeśli mamy tak działający kod, to mówimy, że działa **synchronicznie**. Synchroniczność to wykonywanie zadań po kolei i czekania na zakończenie zadania, by przejść do kolejnego.

# Synchronicznie/Asynchronicznie

Czasami taki ograniczenie (blokowanie wykonywania dalszego kodu, kolejnych zadań), z różnych powodów, np. oczekiwanie na informację zwrotną (pobranie danych), nie jest pożądane. Wtedy do gry wchodzi **asynchroniczność**, będąca przeciwieństwem synchroniczności.

Asynchronicznie oznacza, że program **nie robi czegoś od razu, a będzie coś robił w przyszłości**, dzięki czemu **nie następuje blokowania programu**.

Działają tak np. metody związane z AJAX (pierwsza litera znaczy właśnie "asynchroniczny"), eventy czy właśnie metoda setState w React (dlatego ten temat tu poruszamy :) ).



# Działanie synchroniczne w praktyce

zadanie1

-> wykonuję

-> wykonałem

zadanie2

-> pobierz dane

... oczekiwanie na pobranie -> pobrałem dane

-> wyświetl dane

zadanie3

-> wykonuję

-> wykonałem

# Działanie asynchroniczne w praktyce

zadanie1

-> wykonuję

-> wykonałem

zadanie2

-> pobierz dane (asynchroniczna - nie blokujemy)

zadanie3

-> wykonuję

-> wykonałem

...zadanie4, zadanie5 ...

*... w przyszłości, nie wiemy kiedy dokładnie np. po wielu innych zadaniach, pobranie danych się zakończy (sukcesem czy nie)*

-> dane wskazane w zadaniu2 zostały pobrane (sukces, ale może być też porażka). Wywołuję zadanie wyświetlenia danych, zadanie **ustawia się w kolejce** do wykonania

# setTimeout i wywołanie zwrotne (callback)

```
setTimeout(() => console.log("minęło 10 sekunda"), 10000)
```

TERAZ --> PÓŹNIEJ

Metoda asynchroniczna, nie wstrzymuje działania programu. Program nie zatrzymuje się na 10 sekund (jak w tym przykładzie)

Tekst zostanie wyświetlony w konsoli po co najmniej 10 sekundach, tzn. funkcja callback (wywołanie zwrotne) ustawi się w kolejce do wykonania (w pętli zdarzeń) po 10 sekundach.

# Program działa dalej

```
setTimeout(() => this.counter++, 2000)
```

```
setTimeout(() => console.log(this.counter), 1000) //0
```

```
console.log(this.counter) //0
```

# Pętla zdarzeń


```
let number = 0  
setTimeout(() => number++, 0)  
console.log(number)
```

Samo wywołanie asynchroniczne (nawet jeśli opóźnienie wskażemy na 0, powoduje, że ustawia się na końcu kolejki (stosu) do wykonania.

# Pętla zdarzeń

```
let number = 0
setTimeout(() => number++, 0)
console.log(number) //0
```

Zadania do wykonania (w uproszczeniu by lepiej zrozumieć)

- utwórz zmienną number i przypisz 0
- wywołaj funkcję setTimeout
- minęło 0, ustaw na końcu kolejki zdarzeń wywołanie funkcji ()=> number++
- pokaż w konsoli  zawartość zmiennej number
- (dodane na koniec kolejki) wywołaj funkcję ()=> number++

# setState - asynchroniczność

setState, to metoda wywoływana asynchroniczna, oznacza to, że nie jest wywoływana od razu, więc, co bardzo istotne, **zmiana zawartości obiektu state nie jest natychmiastowa**. Jednak zmiana (wywołanie setState) nastąpi przed wywołaniem metody render, więc tam będą już aktualne dane (aktualny state).

Twórcy react wybrali to rozwiązanie ze względów optymalizacyjnych oraz ze względu na rozwiązania, które być może pojawią się w przyszłości. Developerom React pozostaje tylko o tym pamiętać.

Zapamiętaj setState jest metodą wywoływaną asynchronicznie (wewnątrz metod obsługujących zdarzenie w React). **Oznacza to, że React opóźni wykonanie metody setState do czasu wykonania metody odpowiedzialnej za obsługę zdarzenia. Na pewno jednak setState zaktualizuje stan przed metodą render.**

# setState - asynchroniczność

```
state = { counter: 0 }

handleClick = () => {
  this.setState({
    counter: this.state.counter + 1
  })
  /... kod .../
  console.log(this.state.counter);
}
```

1. klik -> zdarzenie kliknięcia (zakładam, że jest gdzieś ustawiony event onClick)
2. uruchomienia metody (w przykładzie handleClick)



# setState - asynchroniczność

```
state = { counter: 0 }

handleClick = () => {
  this.setState({
    counter: this.state.counter + 1
  })
  /... kod .../
  console.log(this.state.counter);
}
```

1. klik -> zdarzenie kliknięcia (zakładam, że jest gdzieś ustawiony event onClick)
2. uruchomienia metody (w przykładzie handleClick)
3. metoda setState będzie wywołana asynchronicznie (po zakończeniu metody handleClick)
4. w konsoli pokazuje się wartość 0
5. metoda handleClick została wykonana.
6. o ile nie ma już nic w kolejce pętla zdarzeń wykonuje setState (tutaj jeszcze dochodzi do scalania tych metod, gdy jest więcej niż jedna, o czym w przyszłości)
7. wywołanie setState powoduje aktualizację komponentu (przede wszystkim wywołujemy metodę render w komponencie).

# setState - scalanie

Jeśli w metodzie obsługującej zdarzenie wystąpi wiele metod setState to **metody są scalane**, tak by nastąpiło tylko **jedno renderowanie** (jest to element optymalizacji).

Scalenie może powodować problem, polegający na powstaniu konfliktu gdy przekazujemy kilka wartości dla tej samej właściwości.

# setState - scalanie, problem

```
state = { counter: 0 }
```

```
handleClick = () => {  
  this.setState({  
    counter: this.state.counter + 2  
  })  
  this.setState({  
    counter: this.state.counter - 2  
  })  
}
```

// Jaka będzie wartość counter w chwili wywołania metody render?

# setState - scalanie, problem

```
state = { counter: 0 }

handleClick = () => {
  this.setState({
    counter: this.state.counter + 2
  })
  this.setState({
    counter: this.state.counter - 2
  })
}
// wartość counter to -2
```

# setState - co będzie? [asynchroniczne wywołanie]

```
state = {  
  number: 0  
}  
  
handleIncrease = () => {  
  this.setState({  
    number: this.state.number + 1  
  })  
  console.log(this.state.number); // co się wyświetli?  
}
```

# setState - co będzie?

```
state = {  
  number: 0  
}  
  
handleIncrease = () => {  
  this.setState({  
    number: this.state.number + 1  
  })  
  console.log(this.state.number); // 0  
  //Wyjaśnienie: wywołanie asynchroniczne, nie wywołuje się od  
  razu (a jedynie ustawia do stosu zdarzeń)  
}
```

# setState - co będzie? [asynchroniczne wywołanie]

```
state = {  
  number: 0  
}  
  
handleIncrease = () => {  
  this.setState({  
    number: this.state.number + 1  
  })  
  this.setState({  
    number: this.state.number + 2  
  })  
  this.setState({  
    number: this.state.number + 3  
  })  
  console.log(this.state.number); //co się pojawi?  
}
```

# setState - co będzie?

```
state = {  
  number: 0  
}  
  
handleIncrease = () => {  
  this.setState({  
    number: this.state.number + 1  
  })  
  this.setState({  
    number: this.state.number + 2  
  })  
  this.setState({  
    number: this.state.number + 3  
  })  
  console.log(this.state.number); // zobaczmy 0  
}
```



# setState - co będzie?

```
state = {  
  number: 0  
}  
  
handleIncrease = () => {  
  this.setState({  
    number: this.state.number + 1  
  })  
  this.setState({  
    number: this.state.number + 2  
  })  
  this.setState({  
    number: this.state.number + 3  
  })  
  
  console.log(this.state.number); // zobaczymy 0, bo na tym etapie oczywiście setState  
                                   jeszcze się nie wywołało (asynchroniczne wywołanie)  
}
```

# setState - co będzie? [scalanie]

```
state = {  
  number: 0  
}  
  
handleIncrease = () => {  
  this.setState({  
    number: this.state.number + 1  
  })  
  this.setState({  
    number: this.state.number + 2  
  })  
  this.setState({  
    number: this.state.number + 3  
  })  
}
```

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

# setState - co będzie?

```
state = {  
  number: 0  
}  
  
handleIncrease = () => {  
  this.setState({  
    number: this.state.number + 1  
  })  
  this.setState({  
    number: this.state.number + 2  
  })  
  this.setState({  
    number: this.state.number + 3  
  })  
}
```

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

// zobaczymy 3 (a nie 0 czy jak można by sądzić 6). Dzieje się tak ponieważ wywołania są scalane do jednego. Ostatnia właściwość nadpisuje poprzednie (o ile się pojawią). Tak jest rozwiązywany ten konflikt.

# setState - callback


Istnieje rozwiązanie "problemu" z merge (scalaniem, łączeniem). Zamiast przekazywać obiekt możemy przekazać funkcję, która zwraca obiekt.

```
handleIncrease = () => {  
  this.setState(prevState => ({  
    number: prevState.number + 1  
  }))  
  
  this.setState(prevState => ({  
    number: prevState.number + 2  
  }))  
  
  this.setState(prevState => ({  
    number: prevState.number + 3  
  }))  
}
```

# setState - callback

Istnieje rozwiązanie "problemu" z merge (scalaniem, łączeniem). Zamiast przekazywać obiekt możemy przekazać funkcję, która zwraca obiekt.

```
handleIncrease = () => {  
  this.setState(prevState => ({  
    number: prevState.number + 1  
  }))  
  
  this.setState(prevState => ({  
    number: prevState.number + 2  
  }))  
  
  this.setState(prevState => ({  
    number: prevState.number + 3  
  }))  
}
```



Nie przekazujemy już do metody obiektu {}, ale funkcję, która zwraca obiekt. W momencie wywołania funkcji setState mamy sytuację w której każda z tych funkcji jest wywoływana, po kolei (na końcu i tak jest scalanie i zwracany jest jeden obiekt)

# setState - callback nie wpływa na asynchroniczność

Nie rozwiązuje to "problemu" asynchroniczności, co widać w przykładzie poniżej, ale problem scalania (co zobaczymy za chwilę).

```
handleIncrease = () => {  
  this.setState(prevState => ({  
    number: prevState.number + 1  
  }))  
  
  this.setState(prevState => ({  
    number: prevState.number + 2  
  }))  
  
  this.setState(prevState => ({  
    number: prevState.number + 3  
  }))  
  //console.log(this.state.number) oczywiście będzie takie jak bez setState  
  //Czyli zakładając, że w obiekcie state number wynosi 0 na tym etapie też będzie 0  
}
```

# setState - callback wpływa na scalanie

```
handleIncrease = () => {  
  this.setState(prevState => ({  
    number: prevState.number + 1  
  })))  
  
  this.setState(prevState => ({  
    number: prevState.number + 2  
  })))  
  
  this.setState(prevState => ({  
    number: prevState.number + 3  
  })))  
}
```

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

// zobaczymy 6 (a nie 3 jak przy scalaniu, gdy przekazujemy tylko obiekt). Ma miejsce scalanie, ale też łańcuch aktualizacji. Dzięki temu każda kolejna funkcja zwrotna ma aktualne dane ze state (przekazane w pierwszym parametrze), ponieważ poprzednia funkcja zwrotna zwraca aktualny obiekt.

# setState - jak to działa

```
handleIncrease = () => {  
  this.setState(state => ({  
    number: state.number + 1  
  })))  
  
  this.setState(state => ({  
    number: state.number + 2  
  })))  
}
```

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

// aktualizacja stanów związana z wywołaniem setState poprzedza re-renderowania aplikacji.

// ostatnia metoda setState i tak decyduje co będzie zwrócone (bo następuje scalanie). Ważne jest jednak, to, że dzięki wywołaniu zwrotnemu możemy mieć dostęp do wartości obiektu (którym zastąpimy aktualny stan) wynikającego z aktualnych zmian dokonanych przez wywołania zwrotne.



# setState - zrozumieć funkcyjny setState

Pamiętajmy, że na tym etapie `this.state` nie ulega jeszcze zmianie, więc takie skorzystanie jest błędne

```
state = {  
  a: 0,  
  b: 0,  
}  
  
handleClick = () => {  
  this.setState(() => ({  
    a: 5,  
    b: 2,  
  }))  
  this.setState(() => ({  
    a: this.state.a + 1,  
    b: 1,  
  }))  
}  
// {a: 1, b: 1}
```

# setState - zrozumieć funkcyjny setState

Więc korzystajmy z paramtru jeśli coś zmieniamy \*nazwa nie jest istotna, najczęściej używamy state czy prevState

```
state = {  
  a: 0,  
  b: 0,  
}
```

```
handleClick = () => {  
  this.setState(() => ({  
    a: 5,  
    b: 2,  
  })))  
  this.setState(prevState => ({  
    a: prevState.a + 1,  
    b: 1,  
  })))  
}  
// {a: 6, b: 1}
```

# setState - drugi parametr to props

Możemy też przekazać drugi parametr (będzie to props), ale nie musimy (tak jak nie musimy pierwszego)

```
handleIncrease = () => {  
  this.setState((state, props) => ({  
    number: (state.number + 1) * props.multi  
  })))  
  
  this.setState((prevState, props) => ({  
    text: ''  
  })))  
}
```

# setState - zasady scalania

W przypadku konfliktu (setState zmienia w jednej metodzie więcej niż raz dana właściwość) konflikt jest rostrzygnięty w oparciu o najnowsze (najpóźniej występujące w kodzie) ustawienie właściwości.

```
this.setState({  
  a: 1,  
  b: 2  
})
```

```
this.setState({  
  a: 3  
})
```

//scalanie daje taki efekt: {a: 3, b: 2}

# setState - asynchroniczność

Zmiana w obiekcie state nie jest wykonywana od razu, jest wykonywana później (później oznacza niemal od razu, ale nie w metodzie obsługującej zdarzenie). Wynika to z tego, że setState jest wykonywana asynchronicznie.

```
//a: 0, b: 0
```

```
this.setState({  
  a: 5,  
  b: 2,  
})
```

```
this.setState({  
  a: this.state.a + 1 //na tym etapie state ma ciągle takie same właściwości jak na początku  
})
```

```
//scalanie daje taki efekt: {a: 1, b: 2}
```

# setState - jak to działa

Użycie funkcji jako wywołania zwrotnego zapewnia możliwość skorzystania z aktualnego obiektu, który zastąpi stan (na końcu i tak jest scalanie)

```
//a: 0, b: 0
```

```
this.setState(() => ({  
  a: 5,  
  b: 2,  
}))
```

```
this.setState(state => ({  
  a: state.a + 1  
}))
```

// Do funkcji wywołania zwrotnego przekazywany jest argument w postaci aktualnego obiektu, uwzględniającego zmiany

// Do drugiego setState została przekazana jako state (nazwa dowolna) obiekt {a: 5, b: 2}

// Pamiętajmy, że także tu mamy konflikt przy scalaniu i wzięty zostanie na końcu obiekt {a:6, b:2} jako obiekt aktualizujący state.

# setState - jak to działa

Kolejność ma znaczenie

```
//a: 0
```

```
this.setState({  
  a: this.state.a + 5  
})  
this.setState(state => ({  
  a: state.a + 1  
}))
```

//daje taki efekt: {a: 6}

//odwrotna kolejność da {a:5}

# setState - dwa parametry

//nazwy parametrów dowolne

```
handleIncrease = () => {  
  this.setState(prevState => ({  
    number: prevState.number + 1  
  })))  
  
  this.setState(state, props => ({  
    number: state.number + props.value  
  })))  
}
```



# kiedy jakiej składni setState używać?

1. Wystarczy mieć świadomość co robimy i jak działa setState. Wybór sposobu zależy od developera.
2. Jeśli dla ustawienia właściwości używamy odniesienia do poprzedniego stanu (w istocie aktualnego stanu), to lepszym rozwiązaniem może być przekazanie funkcji.
3. Wśród developerów znajdziesz różne przyzwyczajenia. Niektórzy zawsze przekazują funkcje, a inni wybierają ten sposób rzadziej, przekazując zazwyczaj bezpośrednio obiekt.
4. " Functional setState is the future of React"

```
this.setState(prevState => ({  
  number: prevState.number + 1  
}))
```

```
this.setState({  
  number: this.state.number + 1  
})
```

Zobaczmy w kodzie...

# CYKL ŻYCIA

KOMPONENTU

Teoria

# Cykl życia - lifecycle

Komponent (stanowy), to nie tylko **obiekt state** i **metoda render**, ale także inne metody, które możemy (a czasami musimy) wykorzystać pracując z React.

Istnieją różne etapy życia komponentu. Komponent jest **montowany (mounting)**, **aktualizowany (updating)** i **usuwany (unmounting)**. Te etapy też składają się z różnych zdarzeń, które mogą być obsługiwane przez wbudowane metody.

**Metoda render** też jest elementem cyklu życia komponentu (jedynym wymaganym).

# Cykl życia - lifecycle - po co?

M.in.:

Pobranie danych (integracja z API) w inny sposób niż zdarzenie.

Skorzystanie z właściwości DOM (rozmiar, pozycja).

Skorzystanie z bibliotek pracujących z DOM (np. jQuery).

Optymalizacja - np. renderować ponownie czy nie? (domyślnie przy wywołaniu setState, ale niekoniecznie)

# Cykl życia komponentu - 3 fazy

Inicjalizacja (narodziny, tworzenie, montowanie) - [Mounting](#)

//Komponent renderowany pierwszy raz. Definiowane props i state. Komponent główny i jego dzieci są zamontowane na stronie. Możemy też wykonać pewne działania po zamontowaniu. [Tylko raz.](#)

Aktualizacja (wzrost, re-renderowanie) - [Updating](#)

//Komponent renderowany każdy kolejny raz (np. interakcja z UI przez użytkownika). Nowe obiekty props i zmiany w state, ponowne renderowanie komponentów. To ciągły, nieprzerwany proces życia aplikacji (do ewentualnej śmierci).

Usuwanie (śmierć, odmontowanie) - [Unmounting](#)

// Komponent zostaje odmontowany (usunięty), to znaczy nie jest już wyświetlany w interfejsie (np. użytkownik coś wyłączył/schował). W tym cyklu możliwe jest wyczyszczenie aplikacji z rzeczy, które robił komponent, a które nie są już potrzebne po jego usunięciu.

# Komponent klasowy i kolejność

Cykl życia komponenty dotyczy tylko komponentów tworzonych w oparciu o klasę.

Wszystkie metody cyklu życia komponentu będą wywoływane **w określonej kolejności**. Poza metodą render inne metody nie są wymagane i jeśli nie zostaną zdefiniowane, to nie będą wywołane (to nie jest problem)

Cykl życia komponentów rozpoczyna się w raz z wywołaniem `ReactDOM.render()` głównego komponentu.

# Cykl życia - Mounting (1 faza)

`constructor()` - metoda wywoływana przy tworzeniu instancji komponentu (klasy). Nie jest częścią tego cyklu wprost, bo jest cechą klasy JS a nie komponentu, ale jest wywoływana jako pierwsza i dobrze o tym pamiętać.

`componentWillMount()` - nie będzie dostępny od wersji 17 React. Wywoływana tuż przez metodą `render`.

`render()` - zwraca zawartość komponentu (element React) i umieszcza ją na stronie.

`componentDidMount()` - po renderowaniu w interfejsie przeglądarki (powstał DOM). Miejsce np. na pobranie danych.



# Cykl życia - Mounting (1 faza)

```
class Life extends React.Component {  
  constructor(props) {  
    super(props)  
    this.state = {  
      id: 1,  
    }  
  }  
  
  componentDidMount() {}  
  
  render() {  
    return <p>Example</p>  
  }  
}
```

# constructor()

Tylko raz, w chwili tworzenia instancji komponentu.

Używamy do:

- inicjalizacja state (`this.state = {/.../}`)
- wiązanie metod z obiektem (użycie metody `bind`)

Inicjalizacja danych, inicjalizacja obiektu state, czy wiązanie `bind` - można to zrobić bez deklarowania konstruktora.

# render()

Metoda wymagana dla komponentu klasowego.

Tworzy elementy React, które będą umieszczone (zamontowane) w DOM przeglądarki (za pomocą ReactDOM.render)

Tutaj nie zmieniamy state! (nie używamy setState)

Uncaught Error: Maximum update depth exceeded. This can happen when a component repeatedly calls setState inside (...). React limits the number of nested updates to prevent infinite loops.

Metoda render rozpoczyna też cykl życia komponentów dzieci (które są zwracane w metodzie) - oczywiście o ile są to komponenty klasowe (inaczej tylko tworzy).

# componentDidMount()

Wywoływana tylko raz, w chwili, kiedy komponent (i jego dzieci) jest już dodany (zamontowany) w DOM przeglądarki.

Na tym etapie (po metodzie render) nasze elementy React są już renderowane jako węzły (elementy) DOM.

To bardzo przydatna funkcja, kiedy chcemy pobierać dane (np. metoda fetch) do naszego komponentu (możemy już na tym etapie wykonać metodę setState), ustawić setInterval/setTimeout, czy zintegrować z innymi bibliotekami, które pracują na DOM.

# componentDidMount()

Metoda cDM jest wywoływana od razu po tym kiedy komponent został dodany do drzewa DOM (a więc po metodzie render). To właściwe miejsce dla inicjalizacji elementów potrzebnych w węzłach DOM, czy w sytuacji gdy potrzebujemy danych o DOM (np. wielkość czy pozycja).

To też najlepsze miejsce na zadania związane z użyciem bibliotek korzystających z DOM czy związane z użyciem API, tj. pobranie danych ustawienie.

Pamiętajmy, że zmiany za pomocą setState powodują ponowne renderowanie komponentu (wtedy już nie wywoła się componentDidMount a componentDidUpdate)

# Cykl życia - Updating (2 faza)

Ta faza dotyczy komponentów, które są już w DOM i są aktualizowane (czyli domyślnie za każdym razem gdy wywołujemy setState).

render() - tą metodę już znamy, czyli ponowe (przy każdej aktualizacji) wywołanie metody render

`componentDidUpdate()` - po ponownym renderowaniu (tak jak za pierwszym razem mieliśmy `componentDidMount`)

---

*Istnieją też inne, rzadziej używane metody (nie jest ich dużo). Na tym etapie nie ma jednak co zawracać sobie nimi głowy.*

# componentDidUpdate(prevProps, prevState)

Po każdym ponownym renderowaniu (nie przy pierwszym). Dobre miejsce do pobrania nowych danych.

Uwaga na nieskończone pętle.

```
componentDidUpdate() {  
  this.setState({  
    number: this.state.number + 1  
  })  
}
```

Uncaught Error: Maximum update depth exceeded. This can happen when a component repeatedly calls `setState` inside `componentWillUpdate` or `componentDidUpdate`. React limits the number of nested updates to prevent infinite loops.

W takich sytuacjach, trzeba zbudować warunek w którym można skorzystać z przekazanych argumentów.

# Cykl życia - Unmounting (3 faza)

Usunięcie komponentu. Czyścimy naszą aplikację np. wyłączmy setInterval.

```
componentWillUnmount()
```



Przejdźmy do kodu...