Overview training

React + Express

21.11 - 25.11

sages

Główne obszary działania

Szkolenia

Oferujemy szeroki katalog szkoleń z technologii mainstreamowych i specjalistycznych, wschodzących i legacy. Zajęcia prowadzimy w trybie warsztatowym, a programy są oparte o praktyczne know-how. Specjalizujemy się w prowadzeniu dedykowanych szkoleń technologicznych, których agendę dostosowujemy do potrzeb naszych klientów i oczekiwań uczestników.

Kursy rozwojowe

Posiadamy kursy otwarte (Kodołamacz.pl) i dedykowane akademie dla firm, pozwalające na zdobycie nowych kompetencji w ramach kompleksowych programów rozwojowych dla pracowników. Oferujemy również wsparcie w rekrutacji i edukacji przyszłych pracowników naszych klientów.

E-learning połączony z warsztatami

Jako uzupełnienie szkoleń tradycyjnych oraz formę nauki samodzielnej proponujemy kursy e-learningowe. Aktualnie w naszej ofercie dostępne są szkolenia typu masterclass, rozumiane jako szkolenia wideo, uzupełnione o spotkania/warsztaty na żywo (zdalnie) z autorem kursu.

Studia podyplomowe

Współpracujemy z uczelniami wyższymi wspierając realizację zaawansowanych kierunków studiów z zakresu specjalistyki IT. Jesteśmy partnerami studiów podyplomowych:

na Politechnice Warszawskiej:
Data Science: Algorytmy, narzędzia i aplikacje dla problemów typu Big Data
Big Data: Przetwarzanie i analiza dużych zbiorów danych
Wizualna analityka danych.

na Akademii Leona Koźmińskiego:
Data Science i Big Data w zarządzaniu
Chmura obliczeniowa w zarządzaniu projektami i organizacją.

Wydarzenia IT

Inspirujemy i szerzymy wiedzę o technologiach z różnych obszarów na kilkugodzinnych, praktycznych warsztatach w ramach naszej inicjatywy Stacja IT. Organizujemy konferencje m.in. AI & NLP Day, Testaton. Występujemy na konferencjach wewnętrznych naszych klientów np. Orange Developer Day.

sages

Who am I?

Mateusz Jabłoński

- programmer since 2011
- React / Angular on Frontend
- NodeJS / Java on Backened
- sometimes blogger / trainer / mentor

mateuszjablonski.com
mail@mateuszjablonski.com

From the beginning

The arrangements

- our goal and agenda of workshops
- mutual expectations
- questions and discussions
- flexibility
- openness and honesty

What awaits us?

Day 1.

- React
  - Architecture
  - Idea around the Virtual DOM
  - Components
  - Object programming in React
  - Functional programming in React
  - State management
  - Events
  - Life cycle methods
  - React Hooks

What awaits us?

Day 2.

- React in practice
  - Create React App
  - Complex views
  - Typing components
  - Styling components
    - CSS Modules
    - CSS in JS
    - Styled Components
  - Forms
  - Routing
  - Communication with API
  - Envs in React
  - React Portal
- SSG in Gatsby
- SSR in NextJS
- SSR in React
- React 18 - what’s new?
- Server components

What awaits us?

Day 3.

- State management
  - Context API
  - FLUX
  - Redux
  - Mobx
- Programming technics
  - MPA
  - SPA
  - SSR
  - SSG

- NodeJS
  - CommonJS
  - NodeJS as a server
  - Global objects in Node
  - Modules
  - File System

What awaits us?

Day 4.

- ExpressJS
  - Introduction
  - Installation and configuration
  - Project structure
  - Routing
  - Communication
  - Error handling
  - HTML in ExpressJS
  - Response and Request
  - Templates engines
  - Complex views: Partial, Layout, Blocks
  - Forms
  - Validation
  - Upload
- Authorization and authentication
  - Cookies and user session
  - Register and login
  - Middlewares
- Databases
- Rest API
- Swagger
- SocketIO

What awaits us?

Day 5.

- Testing
  - Definition and scope of unit tests
  - What and how test?
  - Black box vs. white box
  - Possibilities of React Testing Utilities
  - Testing React components
  - Events in tests
  - Shallow / full rendering
  - Contract testing
- Storybook
- Performance
- Proxy

github.com/matwjablonski/bsh-training

React

Babel

Babel in React

- Babel is one of the basic tools used in React app development process
- Configuration is kept in .babelrc file
- Configuration contains two groups of rules:
  - presets - rules of code transformations
  - plugins - allows extend Babel engine

```
{
  "presets": ["@babel/preset-react"],
  "plugins": ["react-hot-loader/babel"]
}
```

React

Basics


React

React

- React is a library which can be used for building dynamic, complex user interfaces in declarative and module way
- It takes off responsibility for rendering and updating state of DOM tree from programmer
- Allows declaring structure and logic of displaying content in more declarative way (more natural than in object oriented methods or in imperative programming)
- Allows to hold structure and logic in one place (**only with Javascript** - programmer can use all possibilities of Javascript, without touching and learning syntax of markup languages)

!

React doesn’t render changes directly,
but through Virtual-DOM

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
    <!-- Load React. -->
    <!-- Note: when deploying, replace "development.js" with "production.min.js".
-->
    <script src="https://unpkg.com/react@18/umd/react.development.js"
crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
crossorigin></script>

    <!-- Load our React component. -->
    <script src="like_button.js"></script>

  </body>
</body>
</html>
```

React

Virtual DOM

Virtual DOM

- VirtualDOM is simplified representation of DOM (Document Object Model) objects, which are in the browser. Rendering cycle looks like:
  - React will build VirtualDOM tree in a declarative way, in the next step render it as a browser DOM
  - In case of state change React will build new Virtual DOM tree of changed component
  - ReactDOM is comparing actual and new tree, after that it updates only those places, which really need to be change

Due to the small number of DOM operations, React updates the view instantly!

Virtual DOM

- Allows to generate abstract DOM, which can be rendered as UI on a lot of platforms:
  - **react-dom** - browser DOM
  - React Native - native apps for iOS and Android
  - react-blessed - terminal
  - react-canvas - HTML Canvas element
  - react-vr / react 360 - 3D applications.

React

Attributes and classes

```
React.createElement(
  'div',
  {
    id: 'root_element',
    className: 'root-element-class',
    style: {
      borderTop: '1px solid black',
    }
  },
  "Hello!"
)
```

React

JSX

```
const Section = <section>
  <h1>title</h1>
  <h2 className="subtitle" id="hook_to_title">daily</h2>
  <ul>
    <li>ts</li>
    <li>react</li>
  </ul>
</section>;

React.render(Section, document.getElementById('app'));
```

```
const section = {
  title: 'title',
  subtitle: 'daily',
}

const Section = <section>
  <h1>{section.title}</h1>
  <h2 className="subtitle" id="hook_to_title">{section.subtitle}</h2>
  <ul>
    <li>ts</li>
    <li>react</li>
  </ul>
</section>;

React.render(Section, document.getElementById('app'));
```

```
const section = {
  title: 'title',
  subtitle: 'daily',
  items: [
    {
      id: 1,
      name: 'ts',
    },
    {
      id: 2,
      name: 'react',
    },
  ]
};

const Section = <section>
  <h1>{section.title}</h1>
  <h2 className="subtitle" id="hook_to_title">{section.subtitle}</h2>
  <ul>
    {section.items.map(item => <li key={item.id}>{item.name}</li>)}
  </ul>
</section>;

React.render(Section, document.getElementById('app'));
```

React

Components

Components

- Basic block of React application is Component
- Component is an element in DOM tree, which is manage by React
- **Function / Class of the component** contains Javascript code which controls look and behavior of the element.
- Instance of component is an element, which had been re-render by React in usage of its class or function.
- Components can be nested in any structures, same as HTML or XML
- We can pass to component data as attributes same as it is in HTML. In contrast to HTML we can pass any objects (not only strings). Passed params are known as **component properties** (in short: **props**)

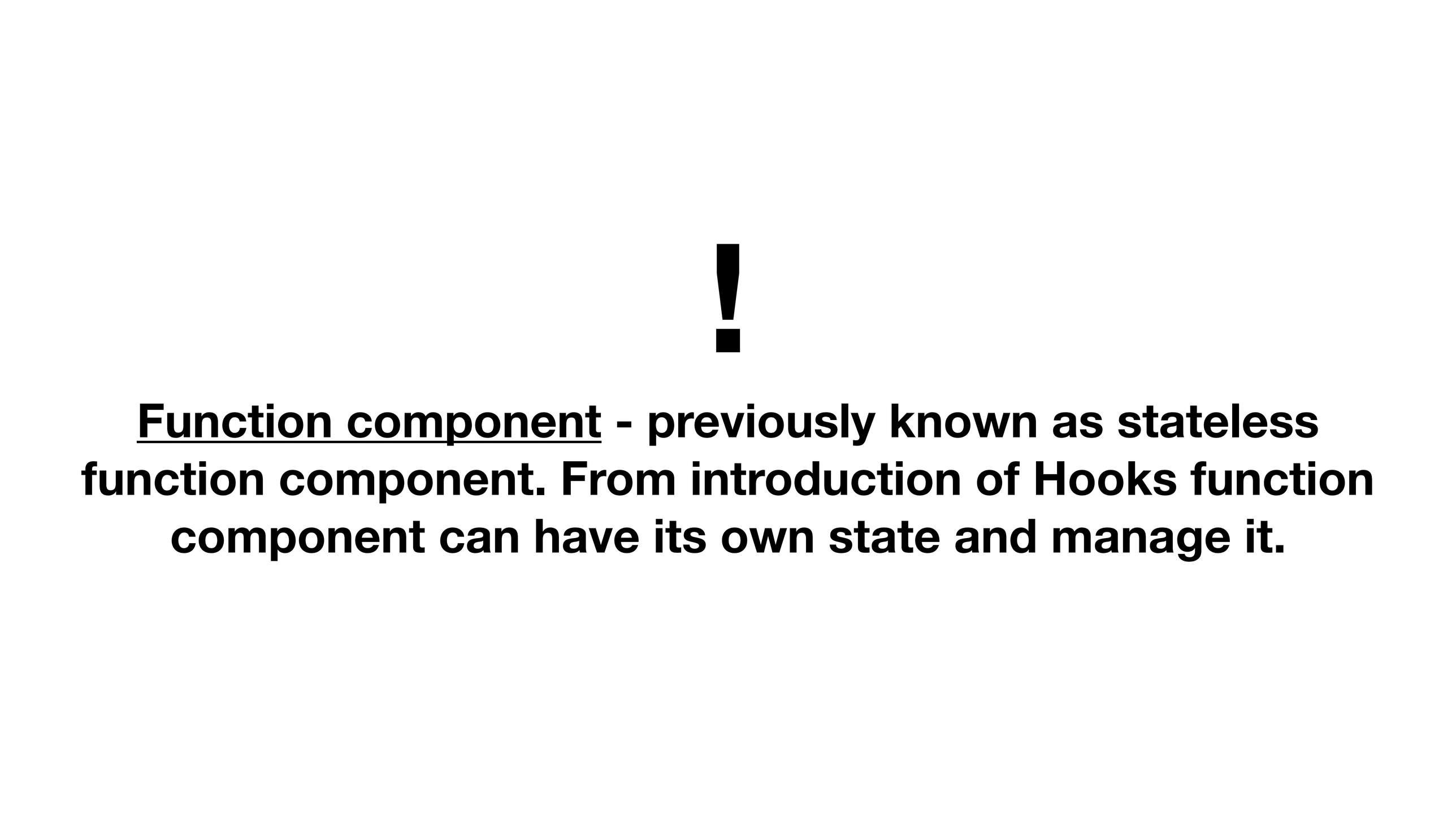
```
<MyComponent option={variable} title="text">
  Text or other components
</MyComponent>
```

```
// Pseudoclass ES6

var createReactClass = require('create-react-class');
var Greetings = createReactClass({
  render: function() {
    return React.createElement('h1', {}, ,Hello,' + this.props.name)
    // return <h1>Hello, {this.props.name}</h1>
  }
});
```

```
// ES6 class

class Greetings extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

```tsx
// TS class

interface GreetingsProps {
  name: string;
}

class Greetings extends React.Component<GreetingsProps> {
  public render(): ReactNode {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

```
// function component

const Greetings = ({ name }) => {

  return <h1>Hello, {name}</h1>
}
```


!
Function component - previously known as stateless function component. From introduction of Hooks function component can have its own state and manage it.

```tsx
// function component in ts

interface GreetingsProps {
  name: string;
}

const Greetings = ({ name }: GreetingsProps) => {

  return <h1>Hello, {name}</h1>
}

const Greetings2: React.FC<GreetingsProps> = ({ name }) => {

  return <h1>Hello, {name}</h1>
}
```

React

Diffing Algorithm

Elements Of Different Types

- In diffing process React takes two trees and check at the beginning types of the root elements
  - When they are different - React purge old tree and create new based on Virtual DOM
  - While tearing down:
    - DOM elements are destroyed
    - Component instances will receive **componentWillUnmount** event (lifecycle method)
    - New tree will be rendered in place of old one
    - New components will receive information that components are mounted

DOM Elements Of Same Type

- When they are same:
  - React takes arguments of both and compare them and in next step it update only those who needs to be updated, like **styles**, **className** etc.
  - Without destroying tree it goes to children and start diffing on children elements.

Components Elements Of Same Type

- When they are same:
  - Between re-renders the instance of component stays the same
  - Component will be informed that something changed and it should re-render itself

Recursing on children

- By default React iterates over lists of children at the same time and generate mutations whenever there is a difference
- If React found anything new in the end of the list it will just add it to the list
- Worse if we are adding item to the beginning of the list (it has worst performance because every next item will be different, so React need to update whole list)
  - To handle that situation we have **key** attribute

Key

- Element in collection should have an **key** attribute.
- Key should be unique and stable between every component render. Due to that React can update lists in most optimal way, avoiding unnecessary re-renders, using as low number of operation as possible on DOM.
- If key will have new value, React re-render whole list once again.
- For the DOM updates is responsible Diffing Algorithm. Process of DOM update is reconciliation.

!

Diffing algorithm is just an implementation detail. We can avoid using it. Then whole app will be re-render on every change.

React

Create React App

```
{
  "name": "test2",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.16.5",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

Task

25 minutes
materials/TASK_14.md

React

State and Props

State and Props

- React „reacts” on data changes and in case of change execute next render of VirtualDOM.
- Rendering can be executed in two scenarios:
  - When component received new properties (**props**)
  - When internal state of component had changed (**state**)
- Passing new properties to element or change of its internal state case re-render of component.
- We shouldn’t update manually props, either state

```
import React from 'react';

const Todo = props => (
  <div>
    <h3>{props.title}</h3>
  </div>
);

const App = () => (
  <Todo title="Say hello” />
)
```

```
import React from 'react';

interface TodoProps {
  title: string;
}

class Todo extends React.Component<TodoProps> {
  render() {
    return (
      <div>
        <h3>{this.props.title}</h3>
      </div>
    )
  }
}

class App extends React.Component {
  render() {
    return (<Todo title="Say hello” />)
  }
}
```

```
import React, { Component } from 'react';

function Clock(props) {
  return <h3>{props.name}</h3>
}

Clock.defaultProps = {
  name: 'Hello'
}
```

```
import React, { Component } from 'react';

class Clock extends Component {
  // default props
  static defaultProps = {
    name: 'Hello',
  }

  constructor(props) {
    super(props);

    // default state
    this.state = { date: new Date() }
  }
}
```

Updating state

- React have to know, when state of component changed. **Don’t modify state directly.**
- For changing state we should use function: **setState** or hook **useState**.
- State methods work asynchronously.
- When state is dependant on previous state we should use callback as first argument of changing state method.

- **Class Way:**
  - Changes are merged (when changing only one value of state object, previous stays the same)
  - Using value of state just after update should cause an error (component could not re-render in time)
  - **this.replaceState({})** - instantly overwrite whole **this.state** object

- **Hook Way:**
  - Changes are overwrited

```
this.setState((prevState, props) => {
  return {
    counter: prevState + props.increment,
  }
})
```

```
interface AProps {
}
interface AState {
}
class A extends React.Component<AProps, AState> {
}
```

```
class Comp extends React.Component {

}

Comp.propTypes = {
  name: PropTypes.string,
}
```

Events

- On events in React can be listen directly on rendered elements
- We can manage on them in components code, object of event is passed as param (prop)
- We can execute code directly

```
const Input = () => {
  const [value, setValue] = useState()

  return (
    <input value={value} onChange={(e: SyntheticEvent) => setValue(e.target.value)}
  )
}
```

```
class Input extends React.Component {
  handleChange(e) {
    this.setState({value: e.target.value})
  }

  render() {
    return (
      <input value={this.state.value} onChange={this.handleChange} />
    )
  }
}
```

Forms

- By adding to form field an value attribute, field is changing into **controlled component**
- **Controlled component** means that value of it, is controlled by React component (in most cases its connected with state of component)
- When field is controlled component only component can change it (on UI input wouldn't react without implementation of change functionality)
- By default all fields are uncontrolled components (browser is responsible for changing value inside inputs) (without attribute value)
- Code `<input value={this.state.myValue} />` makes field impossible to change manually
- To change value of controlled field we should use one of two methods: **setState** or **useState**
- Elements of type **<select>** or **<textarea>** also have attribute **value**

Task

25 minutes
materials/TASK_15.md

Components nesting

- Components can be nested
- Any code which is JSX Element passed between open and close tags of component will be accessible as JSX Object as variable **props.children**
- We can nest also **strings**, **arrays** of **jsx elements** and everything which fulfills type **ReactNode**

```jsx
const LastItem = () => <p>I am last!</p>

const List = ({ data, children }) => {
  return (
    <ul>
      {data.map(item => <li key={item.id}>{item.title}</li>)}
      {children}
    </ul>
  )
}

const App = () => (
  <List data={[]}>
    <LastItem />
  </List>
)

export default List;
```

!

Prop drilling (also: threading) its a process of passing data by props to the bottom of components tree to target component.

React

Lifecycle methods

Life cycle of component

- Every component has implemented life cycle process.
- Life cycle contains methods which are stages of existence component in React VirtualDOM structure.
- Those stages represents different points in life of component from mount to unmount (remove from UI).
- Mounting:
  - **UNSAFE_componentWillMount()** - before mounting in DOM
  - **componentDidMount()** - after mounting in DOM
- Updating:
    - **UNSAFE_componentWillReceiveProps(newProps)** - component will receive new props
    - **getDerivedStateFromProps(newProps)** - component received new props
    - **shouldComponentUpdate(newProps, newState)** - If return false, React omit next render
    - **UNSAFE_componentWillUpdate()** - component will be render, don’t change the state
    - **componentDidUpdate()** - component is rendered, DOM is stable
    - **getSnapshotBeforeUpdate(prevProps, prevState)** - just before update browser DOM

Life cycle of component

- Unmounting:
  - **componentWillUnmount()** - before remove component from DOM
- Exceptions:
  - Methods are executed in case of exceptions while rendering process, in lifecycle methods or constructor methods of child components
  - **getDerivedStateFromError(newProps)**
  - **componentDidCatch()**

React

Operating on DOM

Its not advisable manipulating DOM generated by React, but there is such a possibility. To access DOM elements we can use references.

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.inputRef = React.createRef();
  }

  componentDidMount(): void {
    this.inputRef.current.focus();
  }

  render(): React.ReactNode {
    return (
      <form>
        <input ref="inputRef"/>
      </form>
    )
  }
}
```

```
const Form = ({ data }) => {
  const inputRef = useRef<HTMLInputElement>(null);

  const handleSubmit = (e) => {
    e.preventDefault();

    inputRef.current && inputRef.current.focus();
  }

  return (
    <form onSubmit={handleSubmit}>
      <input ref={inputRef}/>
    </form>
  )
}
```

Dealing with DOM

- To have an access to reference we can use:
  - **ReactDOM.findDOMNode(this.refObj.current);**
  - Above example is kind of emergency exit, docs advise against using that function, because it distorts abstraction of components structure
  - Remember that references will be available only when component is rendered
  - Use references carefully, they should be called only in specific life cycle methods (also keeping data inside references will omit re-render process after change that data)

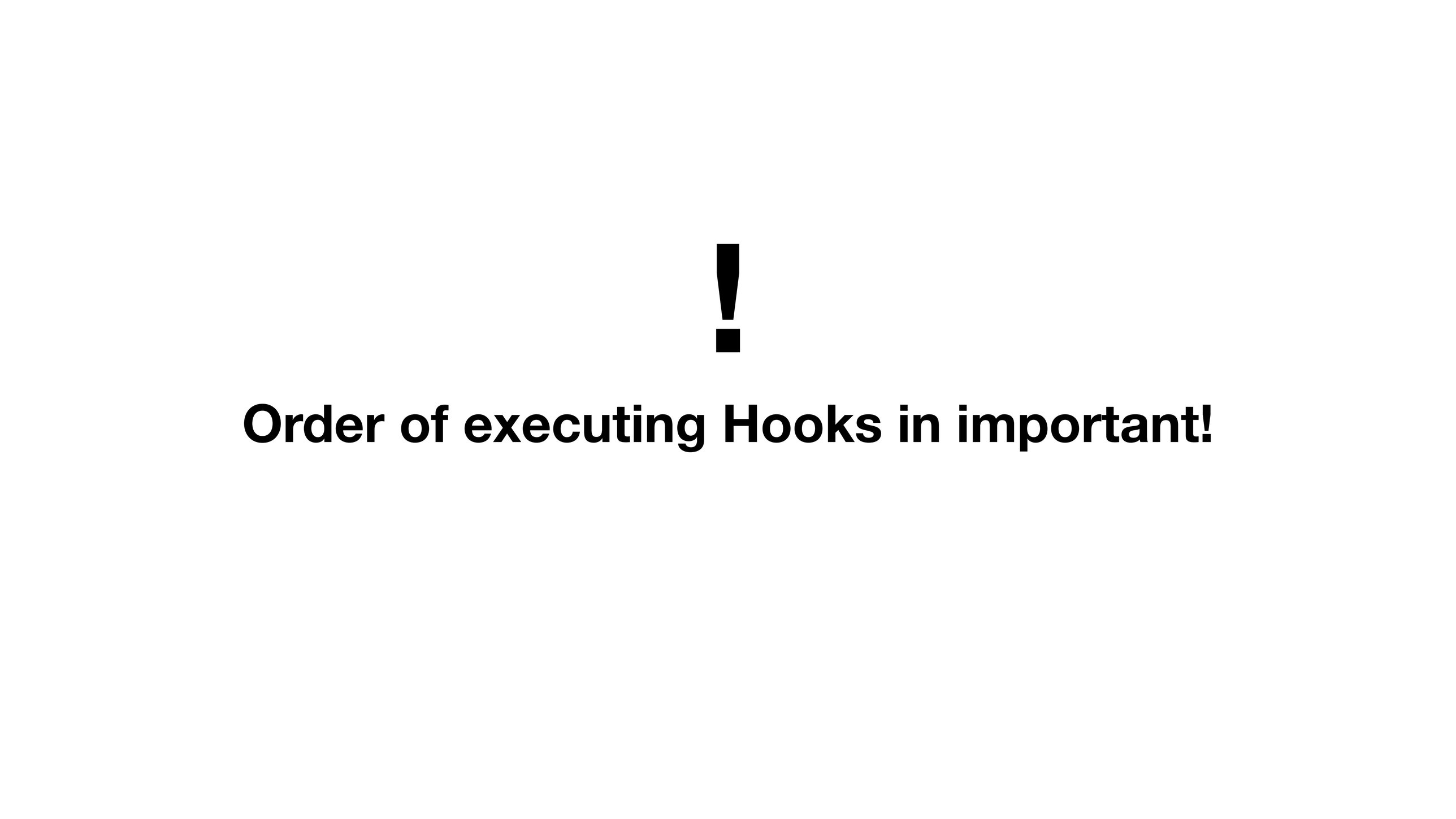
```
return (
      <form>
        <input ref={element => element.focus()}/>
      </form>
    )
```

React

Hooks

React Hooks

- Functions in Javascript doesn’t have state, it means that every execution has access only to it arguments (props in React) and variables from upper scope (closure)
- In React components are functions - so by default they cannot keep state between next calls (re-renders)
- Hooks are functions, which can be places inside component, but only in top scope of component
- Hooks add possibilities to use references, state and side effects inside functional components
- Hooks can be called only in React components
- We can create our own hooks (name of this kind of function should be prefixed by **use** keyword)


!
Order of executing Hooks in important!

```
const Item = ({ data }) => {
  const [ clicked, setClicked ] = useState(0);

  const handleClick = () => {
    setClicked((prevValue) => prevValue + 1);
  }

  return (
    <div onClick={handleClick}>I am clicked {clicked} times</div>
  )
}
```

useState

- It is used for changing state of component
- While calling this hook we can pass default value
- In the component we can have more then one calls of this hook to handle different state for different uses
- **useState** overwrites whole state (in contrast to **setState**)
- Default state can be a function, then while creating state default value will be value returned from this function

Task

25 minutes
materials/TASK_16.md

```
function MyComponent(props) {
  const [state, setState] = React.useState(() => {
    return calculations(props);
  });
}
```

useEffect

- **useEffect** allows to create “side effects”
- After every DOM render we can execute imperative operations on / outside component using hook **useEffect**
- For example we can set title of browser window, fetch data from server / api etc.

```
const WindowTitleHandler = () => {
  const [title, setTitle] = useState('Default title')

  useEffect(() => {
    document.title = title;
  })

  return <div>
    <button onClick={() => setTitle('New title’)}></button>
  </div>
}
```

Every render of component cause new execution of effect. It works similar to life cycle methods: componentDidMount() and componentDidUpdate()

```
const WindowResizeWatcher = () => {
  const [windowWidth, setWindowWidth] = useState(window.innerWidth);

  const handleResizeAction = () => {
    setWindowWidth(window.innerWidth);
  }

  useEffect(() => {
    window.addEventListener('resize', handleResizeAction);

    return () => {
      window.removeEventListener('resize', handleResizeAction);
    }
  })

  return <div>
    Moje okno ma teraz {windowWidth}px szerokości
  </div>
}
```

useEffect

- Second argument of **useEffect** is an array of dependencies (from which depends execution of effect)
  - When second argument is not passed, **useEffect** will be executed after every render
  - When second argument is an empty array, **useEffect** will be executed only once
  - When second argument array contains dependency, **useEffect** will be executed when any of it change

```
useEffect(() => {}, []);
// only after first render

useEffect(() => {});
// after every render

useEffect(() => {}, [ props.title, value ]);
// after change of dependencies
```

useLayoutEffect

- Behavior similar to **useEffect**
- Difference is that it is executed synchronously after mutation in DOM structure
- Useful when we want our view complies with data in situation of mutation DOM structure directly

Other hooki

- useContext
- useRef
- useReducer
- **useMemo** - memoization of values
- **useCallback** - memoization of callbacks (functions)
- **useImperativeHandle** - customizes the instance value that is exposed to parent components when using ref, similar to useRef, but gives control over the value that is returned and allows to replace native functions like blur, focus with custom implementations
- **useDebugValue** - can be used to display a label for custom hooks in React DevTools
- **useId** - generates unique ID, which is stable between every render
- **useTransition** - we can specify, which state update should have bigger priority
- **useDefferedValue** - can show old value, before new will be ready (usefull when view depends on values from external source, like libraries, other components)

Task

25 minutes
materials/TASK_17.md

React

Communication with API

```
const List = (props: any) => {
  const [data, setData] = useState([]);

  const fetchData = async () => {
    const res = await fetch('//example.com/api/data');
    const data = await res.json();

    setData(data);
  }

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

  return <div>
    {data.map(item => <ListItem />)}
  </div>
}
```

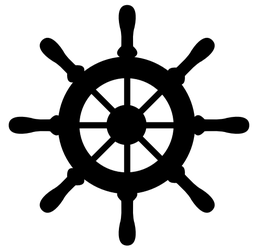
```
const [code, setCode] = useState([]);
  const [isLoading, setIsLoading] = useState(false);

  const fetchCode = () => {
    setIsLoading(true);
    fetch('//example.com/api/data')
      .then(res => res.json())
      .then(data => {
        if (data.status === 404) {
          throw new Error()
        }

        setCode(data);
      })
      .catch((err) => {
        // handle error
      })
      .finally(() => {
        setIsLoading(false);
      })
  }
```

```
const [isLoading, setIsLoading] = useState(false);

const fetchData = async () => {
  setIsLoading(true);
  try {
    const res = await fetch('//example.com/api/data');
    const data = await res.json();

    if (data.status === 404) {
      throw new Error()
    }

    setData(data);
  } catch(err) {

  } finally {
    setIsLoading(false);
  }
}

useEffect(() => {
  fetchData();
}, [])
```

Dodatkowe pytania?

kahoot.it

XXXXX



Ankieta

tinyurl.com/2p8wpe8w

mail@mateuszjablonski.com
mateuszjablonski.com
linkedin.com/in/mateusz-jablonski/

Dziękuję za uwagę


sages