1. Komponenty

a) tworzony przez ciebie komponent musi zaczynać się z wielkiej litery i zwracać HTML container, domyślne kontenery (podstawowe) zaczynają się z małych liter

b) komponent to funkcja w jsx (js)

c) Shift + Alt + F – formatowanie kodu

d) używając komponentów wrapujemy je w <>, np. używamy funkcji Header

```
<Header></Header>
```

e) można szybko dać <Header />

f) rozszerzenie pliku to .jsx (mogą też być .js)

g) renderowanie componentu

```
import ReactDOM from "react-dom/client";

import App from "./App.jsx";
import "./index.css";

const entryPoint = document.getElementById("root");
ReactDOM.createRoot(entryPoint).render(<App />);
```

ReactDOOM tworzy korzeń na podstawie pobranego elementu i wstrzykuje do niego nasz komponent

2. Zmienne dynamiczne

a) wstawiasz zmienne opatulając {zmienna}, może to być wynik funkcji. Nie może to być if, loop, tylko wartość

b) można importować w następujący sposób

```
import reactImg from './assets/react-core-concepts.png';
```

Potem wstawiasz w klamry {}

3. Rekwizyty (props) (Wstawianie danych do komponentu)

a) tworzysz komponent, który przyjmie obiekt

```jsx
function CoreConcept(props) {
  return (
    <li>
      <img src={props.image} alt={props.title} />
      <h3>{props.title}</h3>
      <p>{props.description}</p>
    </li>
  );
}
```

b) wywołując komponent przekazujesz parametry

```jsx
<CoreConcept
  title="Components"
  description="The core UI building block."
  image={componentsImg}
/>
```

c) Możemy też zaimportować inny komponent (rozszerzenie .js)

```jsx
import { CORE_CONCEPTS } from "./data";
```

Opatulamy w {} bo nie jest default export

d) operator rozprzestrzeniania

```jsx
<CoreConcept
  title={CORE_CONCEPTS[1].title}
  description={CORE_CONCEPTS[1].description}
  image={CORE_CONCEPTS[1].image}
/>
```

To samo co

```jsx
<CoreConcept {...CORE_CONCEPTS[1]}/>
```

e) można też

```jsx
function CoreConcept({image, title, description}) {
  return (
    <li>
      <img src={image} alt={title} />
      <h3>{title}</h3>
      <p>{description}</p>
    </li>
  );
}
```

4. Dekompozycja komponentów

a) komponenty zazwyczaj tworzymy w folderze o nazwie components

b) nazwa pliku to zazwyczaj nazwa komponentu

c) musimy użyć: export default (lub export wtedy importując dokładamy {})

d) css także dekomponujemy. Dodajemy pliki css do folderu components

importujemy w potrzebnym komponencie

```jsx
import './Header.css';
```

e) przy dekompozycji css, będzie on i tak stosowany do każdego elementu, wystarczy, że go raz zaimportujemy i będzie działał na każdym komponencie

f) przekazywanie danych. Tworząc komponent z parametrami props, zawsze otrzymasz parametr children

```jsx
<TabButton>Components</TabButton>
```

Chcąc wykorzystać Components musisz zrobić coś takiego (może to być i jsx, tzn. divy, headery itd.)

```jsx
export default function TabButton(props){
    return <li><button>{props.children}</button></li>
}
```

Takie coś nazywamy kompozycją komponentów

5) Obsługa przycisków

a) Ctrl + Space – podpowiedzi

b) dodajemy Onclick parametr, przyjujący funkcję, zazwyczaj robimy to wewnątrz komponentu

c) nazywamy od handle lub kończymy (taka konwencja) i dodajemy nazwę wydarzenia

```
export default function TabButton({ children }) {
    function handleClick() { }
        console.log('Hello word');
    return (
        <li>
            <button onClick={handleClick}>{children}</button>
        </li>
    );
```

d) jeśli coś ma się stać po wykonaniu danego zdarzenia, to nazwę parametru zaczynamy od on (taka konwencja)

e) przekazywanie funkcji jako parametr do przycisku:

i) określasz funkcję w komponencie

```
function handleSelect() {
  console.log("Hello word - 1");
}
```

ii) przekazujesz jako parametr do innego komponent

```
<TabButton onSelect={handleSelect}>Components</TabButton>
```

iii) w niestandardowym komponencie przekazujesz zmienną do natywnego komponentu

```jsx
export default function TabButton({ children, onSelect }) {

  return (
    <li>
      <button onClick={onSelect}>{children}</button>
    </li>
  );
}
```

f) chcąc przekazać parametr do funkcji onClick robimy sztuczkę (niestandardowy parametr):

i) Dodajemy parametr do funkcji obługującej wydarzenie

```jsx
function handleSelect(selectedButton) {
  console.log("Hello word " + selectedButton);
}
```

ii) tworzymy funkcję, wykonującą inną funkcję, przekazując parametr

```jsx
<TabButton onSelect={() => handleSelect("components")}>Components</TabButton>
```

6) Aktualizacja komponentów

a) Komponenty są renderowane tylko na samym początku, więc jeśli chcemy coś zmienić w komponencie, musimy zmienić stan

b) musimy zaimportować ReactHook (funkcje które zaczynają się od use

```jsx
import { useState } from 'react';
```

Mogą być one wywoływane tylko **wewnątrz komponentów reacta**, lub innych react hook, **na samym początku** (nie może być zagnieżdżone w innej funkcji)

c) używanie hook'a:

i) definiujemy

```jsx
const [ selectedTopic, setSelectedTopic ] = useState('Plese click a button');
```

konwencja, useState przyjmuje wartość domyślną

selectedTopic – przechowuje zmienioną wartość

setSelectedTopic – funkcja aktualizująca zmienną, mówiąca reactowi o aktualizacji komponentu

d) zastosowanie

i) dodajemy plik data.js

```
export const EXAMPLES = {
  components: {
    title: 'Components',
    description:
      'Components are the building
    code: `
function Welcome() {
  return <h1>Hello, World!</h1>;
}`,
  },
  jsx: {
```

ii) importujemy

```
import { EXAMPLES } from "./data.js";
```

iii) odnosimy się do zmiennych ustawianych w event handler

```
<h3>{EXAMPLES[selectedTopic].title}</h3>
```

Należy pamiętać o prawidłowej wartości początkowej

7) Zmienne warunkowe

a) ustawiamy useState() usuwając wartość początkową i dajemy {state ? :}

```
{!selectedTopic ? <p>Please select a topic.</p> : null}
{selectedTopic ? (
  <div id="tab-content">
    <h3>{EXAMPLES[selectedTopic].title}</h3>
    <p>{EXAMPLES[selectedTopic].description}</p>
    <pre>
      <code>{EXAMPLES[selectedTopic].code}</code>
    </pre>
  </div>
) : null}
```

Można połączyć w jedno 😊

b) użycie operatora and (&&)

```
{!selectedTopic && <p>Please select a topic.</p>}
```

c) użycie zmiennej trzymającą kod JSX

i) definicja zmiennej

```
let tabContent = <p>Please select a topic.</p>;

if (selectedTopic) {
  tabContent = (
    <div id="tab-content">
      <h3>{EXAMPLES[selectedTopic].title}</h3>
      <p>{EXAMPLES[selectedTopic].description}</p>
      <pre>
        <code>{EXAMPLES[selectedTopic].code}</code>
      </pre>
    </div>
  );
}
```

ii) wyświetlanie

```
{tabContent}
```

8) Dynamiczna stylizacja

a) podobnie jak z przekazaniem funkcji. Po prostu przekazujemy przy wywołaniu parametr odpowiadający za przekazanie klasy:

i) modyfikujemy klasę przyjmującą

```
export default function TabButton({ children, onSelect, isSelected }) {

  return (
    <li>
      <button className={isSelected ? 'active' : undefined} onClick={onSelect}>{children}</button>
    </li>
  );
}
```

ii) przekazujemy wartość dynamicznie na podstawie stanu

```
<TabButton isSelected = {selectedTopic === 'components'} onSelect={() => handleSelect("components")}>
```

9) Dynamiczne wyświetlanie list

a) tablicę danych można wyświetlić dynamiczne

```
{['Hello', 'World']}
```

```
{[<p>Hello</p>, <p>World</p>]}
```

b) ideą jest mapowanie obiektu na element typu HTML

i) mamy dane

```
export const CORE_CONCEPTS = [
  {
    image: componentsImg,
    title: 'Components',
    description:
      'The core UI building block - compose the user interface by combining multiple components.',
  },
```

ii) musimy je zmapować

```
<ul>
  {CORE_CONCEPTS.map((conceptItem) => (
    <CoreConcept {...conceptItem} />
  ))}
</ul>
```

Jest to równoważne wszystkim powtórzeniom tego

```
<CoreConcept {...CORE_CONCEPTS[1]} />
```

iii) żeby pozbyć się warningu ustawiamy key

```
{CORE_CONCEPTS.map((conceptItem) => (
  <CoreConcept key = {conceptItem.title} {...conceptItem} />
))}
```