

JS

Monday, April 29, 2024 12:20 PM

Objects :

```
29 // return userName + messag
30 // };
31
32 const user = {
33   name: "Max",
34   age: 34,
35   greet() {
36     console.log("Hello!");
37     console.log(this.age);
38   }
39 };
40
41 console.log(user.name);
42 user.greet();
43
```

Classes:

```
class User {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log("Hi!");
  }
}

const user1 = new User("Manuel", 35);
console.log(user1);
```

Array:

```

59 const hobbies = ["Sports", "Cooking", "Reading"];
60 console.log(hobbies[0]);
61
62 hobbies.push("Working");
63 console.log(hobbies);
64
65 const index = hobbies.findIndex((item) => {
66   return item === "Sports";
67 });
68
69 console.log(index);
70

```

Console was cleared
Sports
▶ (4) ["Sports", "Cooking", "Reading", "Working"]
0

This type of function is known as arrow function:

```

const index = hobbies.findIndex((item) => item === "Sports");

console.log(index);

```

Short form of the above code., getting rid of the { and return

Map Method : Allows u to transform every item in the array to a another item. It wont change the default array will make a new array with the mentioned changes

```

const editedHobbies = hobbies.map((item) => item + "!");
console.log(editedHobbies);

```

this makes a new array by adding ! To all the items in the array.

O. To make object from the array we can also use map

```

const editedHobbies = hobbies.map((item) => ({text: item}));
console.log(editedHobbies);

```

Here the {} parenthesis is used outside the curly bracket because then if not used then it would be treated as the parenthesis for the function body and not as a new javascript object returned by the arrow function

Console:

```
▼ (4) [Object, Object,
Object, Object]
  ▼ 0: Object
    text: "Sports"
  ▷ 1: Object
  ▷ 2: Object
  ▷ 3: Object
```

Destructuring Array:

```
const [firstName, lastName] = ["Max", "Schwarzmüller"];

// const firstName = userNameData[0];
// const lastName = userNameData[1];

console.log(firstName);
console.log(lastName);
```

Destructuring Object:

```
const { name: userName, age } = {
  name: "Max",
  age: 34
};

console.log(userName);
console.log(age);

// const name = user.name;
// const age = user.age;
```

Destruction in function Parameter

For example, if a function accepts a parameter that will **contain an object** it can be destructured to "pull out" the object properties and make them available as **locally scoped variables** (i.e., variables only available inside the function body).

Here's an example:

1. **function** storeOrder(order) {
2. **localStorage.setItem('id', order.id);**
3. **localStorage.setItem('currency', order.currency);**
4. }

Instead of accessing the `order` properties via the "dot notation" inside the `storeOrder` function body, you could use destructuring like this:

```
5. function storeOrder({id, currency}) { // destructuring  
6.   localStorage.setItem('id', id);  
7.   localStorage.setItem('currency', currency);  
8. }
```

It's very important to understand, that `storeOrder` still only takes one parameter in this example! It does not accept two parameters. Instead, it's one single parameter - an object which then just is destructured internally.

The function would still be called like this:

```
9. storeOrder({id: 5, currency: 'USD', amount: 15.99}); // one argument / value!
```

Spread operator: used to merge arrays :



Spread operation is also available in objects:



For Loop:



Time out function:

For Time out function we need two attributes, 1st the function to execute and the 2nd the time in milli second.

Here to pass a function in the Timeout function we need to pass it without the parenthesis, then it will be passed as a value and will be executed at the timer.

Else if passed with the parenthesis then it will execute at that moment and the value returned from the function will be go as a input argument for the Time out function.

Now there are three method to make the function, mentioned below.

In last example the function is been defined and will be executed after the timer.



✗

React Basic

29 April 2024 12:28

Components

Why Components?

Reusable Building Blocks
Create **small building blocks** & **compose** the UI from them
If needed: **Reuse** a building block in different parts of the UI (e.g., a reusable button)

Related Code Lives Together
Related HTML & JS (and possibly CSS) code is **stored together**. Since JS influences the output, storing JS + HTML together makes sense

Separation of Concerns
Different components handle different data & logic
Vastly **simplifies** the process of working on complex apps

Component Functions Must Follow Two Rules

Name Starts With Uppercase Character
The function name **must start** with an **uppercase** character
Multi-word names should be written in **PascalCase** (e.g., "MyHeader")
It's **recommended** to pick a name that **describes** the UI building block (e.g., "Header" or "MyHeader")

Returns "Renderable" Content
The function must **return** a value that can be **rendered** ("displayed on screen") by React
In **most** cases: Return **JSX**
Also allowed: string, number, boolean, null, array of allowed values

Custom components:

Built-in Components
Name starts with a **lowercase** character
Only **valid, officially defined** HTML elements are allowed
Are **rendered as DOM nodes** by React (i.e., displayed on the screen)

Custom Components
Name starts with **uppercase** character
Defined by you, wraps built-in or other custom components
React **traverses** the component tree until it has only built-in components left

```
1 function App() {
2   return (
3     <div>
4       <header>
5         
6         <h1>React Essentials</h1>
7         <p>Fundamental React concepts you will need for almost any app you are going to build!</p>
8       </header>
9
10      <main>
11        <h2>Time to get started!</h2>
12      </main>
13    </div>
14  );
15}
16
17
18
19
20 export default App;
```

Here we can get the header section in a function and call that function the main code.

This is known as custom components.

```

function Header() {
  return (
    <header>
      
      <h1>React Essentials</h1>
    </header>
    <p>
      Fundamental React concepts you will need for almost any app you are
      going to build!
    </p>
  );
}

```

```

1 function Header() {
2   return (
3     <header>
4       
5       <h1>React Essentials</h1>
6     </header>
7     Fundamental React concepts you will need for almost any app you are
8     going to build!
9     </p>
10    </header>
11  );
12}
13
14 function App() {
15   return (
16     <div>
17       <Header/>
18       <main>
19         <h2>Time to get started!</h2>
20       </main>
21     );
22}
23

```

Import image and save in a variable in the code

```

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

```

DYNAMIC IMAGE USING THE VARIABLE

```

function Header() {
  const description = reactDescriptions[genRandomInt(2)];
  return (
    <header>
      <img src={reactImg} alt="Stylized atom" />
      <h1>React Essentials</h1>
      <p>
        {description} React concepts you will need for almost any app you are
        going to build!
      </p>
    </header>
  );
}

```

Note : don't add quotes like "[reactImg]"

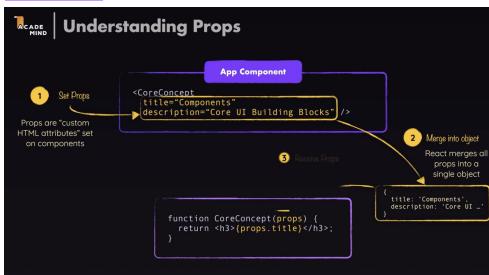
Dynamic value:

```

1 const reactDescriptions = ['fundamental', 'Crucial', 'Core'];
2
3 function genRandomInt(max) {
4   return Math.floor(Math.random() * (max + 1));
5 }
6
7 function Header() {
8   const description = reactDescriptions[genRandomInt(2)];
9
10  return (
11    <header>
12      
13      <h1>React Essentials</h1>
14      <p>
15        {description} React concepts you will need for almost any app you are
16        going to build!
17      </p>
18    </header>
19  );
20}

```

Props concept :



The function `coreConcept` is the base function which is used several time using component in app, and in this we only need to pass props as argument in `coreConcept`, not all the arguments, and then in the main body, we can change the details according to the need.

Note : when calling the function in the body and saving the data in the variable, then we should have the same variable name in the main function..

```

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

function App() {
  return (
    <div>
      <Header />
      <main>
        <section id="core-concepts">
          <h2>Core Concepts</h2>
        </section>
      </main>
    </div>
  );
}

```

```

function App() {
  return (
    <div>
      <Header />
      <main>
        <section id="core-concepts">
          <h2>Core Concepts</h2>
          <ul>
            <CoreConcept
              title="Components"
              description="The core UI building block"
              image={componentsImg}
            />
            <CoreConcept />
            <CoreConcept />
            <CoreConcept />
          </ul>
        </section>
      </main>
    </div>
  );
}

```

```

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

```

Saving image in a variable to use that variable later.

Reduce code in Prop:

```

EXPLORER ... App.jsx • JS data.js U X
REACT-ESSENTIALS
node_modules
src
  assets
    components.png
    config.png
    jsx-ui.png
    react-core-concept...
    state-mgmt.png
  App.jsx
  # index.css
  # index.js
  .gitignore
  index.html
  timeline-lock.json
  TIMELINE
  title={CORE_CONCEPTS[0].title}
  description={CORE_CONCEPTS[0].description}
  image={CORE_CONCEPTS[0].image}
/>
<CoreConcept
  title={CORE_CONCEPTS[0].title}
  description={CORE_CONCEPTS[0].description}
  image={CORE_CONCEPTS[0].image}
/>

```

```

  />
  <CoreConcept {...CORE_CONCEPTS[0]} />
  <CoreConcept

```

```

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

```

Cleaning code:

Previously in the App.jsx file all the components was there, we need to separate all the components into

The screenshot shows two versions of the same code. On the left, the code is expanded to show five individual `<CoreConcept>` components, each with its own title, description, and image props. On the right, the code is reduced to a single `<CoreConcept {...CORE_CONCEPTS[0]} />` call, where the spread operator (`...`) is used to pass all the properties from the array element at index 0 to the component. This demonstrates how the spread operator can be used to reduce repetitive code.

```

... App.jsx • JS data.js U
  ...
  39   <header />
  40   <main>
  41     <section id="core-concepts">
  42       <h2>Core Concepts</h2>
  43       <ul>
  44         <CoreConcept
  45           title={CORE_CONCEPTS[0].title}
  46           description={CORE_CONCEPTS[0].description}
  47           image={CORE_CONCEPTS[0].image}
  48         />
  49         <CoreConcept title="Props" />
  50         <CoreConcept />
  51         <CoreConcept />
  52       </ul>
  53     </section>
  54   </main>
  55 </div>

```

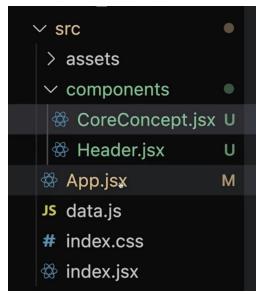
This can be used to fetch all the variable from the array, when the variable name is the same as in the main function.

This reduces the code

Also we can use this syntax to not write `props.title` etc in each line.

its own file

Process :



```
App.jsx Header.jsx

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

```
function App() {
  return (
    <div>
      <Header />
      <main>
        <section id="core-concepts">
          <h2>Core Concepts</h2>
```

We put the header components in a different file and need to use `export default` in front of the function and in the main `App.jsx` file we need to import the header component.

```
import { CORE_CONCEPTS } from './data.js';
import Header from './components/Header.jsx';
```

Also in the header function we are using image which is asset folder outside component folder, so we need to use `../` to get to that folder

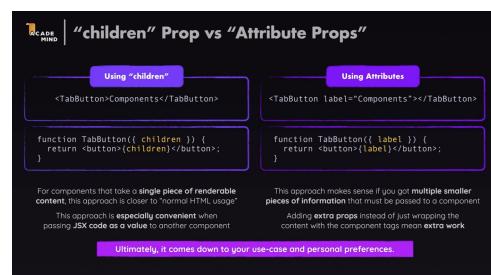
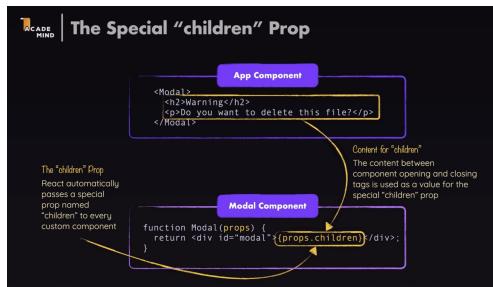
```
import reactImg from '../assets/react-core-concepts.png';
```

NOTE: the name of the file which we are creating should be the same as of the component name. 😊

```
Header.jsx M X # index.css M
1
2
3
4
5
6
7
8
9
10
11
```

Here we are making a `header.css` file and storing the css related to the header in the file and storing it in component folder.

Note: the css wont work until it is imported, so here in the `header.jsx` file we are importing `header.css`



```
</section>
<section id="examples">
  <h2>Examples</h2>
  <menu>
    <TabButton>Components</TabButton>
    <TabButton>JSX</TabButton>
    <TabButton>Props</TabButton>
    <TabButton>State</TabButton>
  </menu>
</section>
```

```
App.jsx M CoreConcept.jsx TabButton.jsx U X
1
2
3
4
5
6
7
8
```

This is known as object destructuring,

The `children` prop is a special prop that's automatically provided to every component function. It contains the wrapped content as a value.

So the `children` prop's value for this code:

1. <Card>
2. <p>Hi there</p>
3. </Card>

would be `<p>Hi there</p>`.

Events:

```
export default function TabButton({ children }) {
  function handleClick() {
    console.log('Hello World!');
  }

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

```
 1 export default function TabButton({ children, onSelect }) {
 2   return (
 3     <li>
 4       <button onClick={onSelect}>{children}</button>
 5     </li>
 6   );
 7 }
```

Explanation:

onSelect goes to tabButton -> onClick triggers onSelect -> handleSelect function runs.

Modifying to generate dynamic output:

```
App.jsx  ●  CoreConcept.jsx  ●  TabButton.jsx  ●

2 import Header from './components/Header/Header.js'
3 import CoreConcept from './components/CoreConcept'
4 import TabButton from './components/TabButton.jsx'
5
6 function App() {
7   function handleSelect() {
8     console.log('Hello World - selected!');
9   }
10 }


```

```
<Section>
<section id="examples">
  <h2>Examples</h2>
  <menu>
    <TabButton onSelect={handleSelect}>Components</TabButton>
    <TabButton onSelect={handleSelect}>JSX</TabButton>
    <TabButton onSelect={handleSelect}>Props</TabButton>
    <TabButton onSelect={handleSelect}>State</TabButton>
  </menu>
```

Now to output different things for different button:-

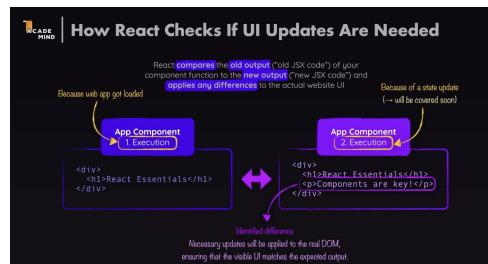
```
function App() {
  function handleSelect(selectedButton) {
    // selectedButton => 'components', 'jsx', 'props', 'state'
    console.log(selectedButton);
  }
}
```

(`) => handleSelect('components')`) is used to avoid the code to trigger when its executed, so the fake function gets executed and the `handleSelect` gets executed when the button is clicked.

```
<menu>
  <TabButton onSelect={() => handleSelect('components')}>
    Components
  </TabButton>
  <TabButton onSelect={() => handleSelect('jsx')}>JSX</TabButton>
  <TabButton onSelect={() => handleSelect('props')}>Props</TabButton>
  <TabButton onSelect={() => handleSelect('state')}>State</TabButton>
</menu>
Dynamic Content
```

By Default, React Components Execute Only Once

You have to “tell” React If A Component Should be Executed Again



Managing State & Using Hooks

-  **Rules of Hooks**
-  **Only call Hooks inside of Component Functions**
React Hooks must not be called outside of React component functions
-  `function App() {
 const [val, setVal] = useState(0);
}` 
`const [val, setVal] = useState(0);
function App() { ... }`
-  **Only call Hooks on the top level**
React Hooks must not be called in nested code statements (e.g. inside of if-statements)
-  `function App() {
 const [val, setVal] = useState(0);
}
if (someCondition){
 const [val, setVal] = useState(0);
}` 

And it will **always** be **exactly** two elements

```
const [stateArray] = useState('Please click a button');
```

Array produced (and returned by React's useState() function)
Contains exactly two elements

The diagram illustrates the useState hook in React. It shows a code snippet where a state variable 'counter' and a function 'setCounter' are returned from the useState hook. The initial state value '0' is stored by React, and the current state value 'Provided by React' is provided by React. A note indicates that the component function will be re-executed if it is rebound again.

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

Initial state value
Stored by React

Current state value
Provided by React

May change if the
component function is
rebound again

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

Function starting with "use" are react hook

Note: when there is a change in the useState (hook function)
Then it will trigger the App component to re-run the component.

useState need a starting value which will show in the page

```

component creation is
executed again
import { useState } from 'react';

function App() {
  const [ selectedTopic, setSelectedTopic ] = useState('Please click
  function handleSelect(selectedButton) {
    // selectedButton => 'components', 'jsx', 'props', 'state'
    setSelectedTopic(selectedButton);
    console.log(selectedTopic);
  }
}

```

```

</menu>
{selectedTopic}
</section>

```

Putting dynamic value, store in data.js file

```

</menu>
<div id="tab-content">
  <h3>{EXAMPLES[selectedTopic].title}</h3>
  <p>{EXAMPLES[selectedTopic].description}</p>
  <pre>
    <code>{EXAMPLES[selectedTopic].code}</code>
  </pre>
</div>
</section>

```

If you want to show the content after the button is clicked then :

```

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

```

2nd approach

```

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

```

3rd approach

```

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>
  );
}

```

```

</menu>
{tabContent}
</section>

```

Examples

- Components
- JSX
- Props
- State

State

State allows React components to change their output over time in response to user actions, network responses, and anything else.

```

function Counter() {
  const [ count, setCount ] = useState(0);
  ...
}

```

Note: when there is a change in the useState (hook function)
Then it will trigger the App component to re-run the component.

useState need a starting value which will show in the page,
And then when triggered, it will return a array with two values
Here we can use array destructuring to get the two value in two different variables.
Here the first value of the array will be the value with the useState is currently storing, it will return that value to the first value of the array.
The 2nd value will be a function provided by react that can be executed to update this state.(i.e it will make the app component to run again.
Therefore we will pass the first array value to get outputed.

To get this styling where its shows which button is selected

```

<menu>
  <TabButton
    isSelected={selectedTopic === 'components'}
    onSelect={() => handleSelect('components')}
  >
    Components
  </TabButton>
  <TabButton
    onSelect={() => handleSelect('jsx')}
  >
    JSX
  </TabButton>
  <TabButton
    onSelect={() => handleSelect('props')}
  >
    Props
  </TabButton>
</menu>

```

State

```
State allows React components to change their output over time in response to user actions, network responses, and anything else.

function Counter() {
  const [isVisible, setIsVisible] = useState(false);
```

We take another attribute in TabButton function and when isSelected is true the class with the css styling gets active.

In the app component, we added a compare to push the isSelected button as true to the tabButton function.

```
<h2>Core Concepts</h2>
<ul>
  {CORE_CONCEPTS.map((conceptItem) => (
    <CoreConcept key={conceptItem.title} {...conceptItem} />
  )));
</ul>
```

Using key to uniquely identify a value.

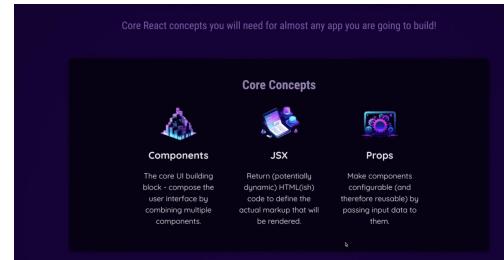
```
1 | import React from 'react';
2 |
3 | import Todo from './Todo';
4 | import './styles.css';
5 |
6 | export const DUMMY_TODOS = [
7 |   'Learn React',
8 |   'Practice React',
9 |   'Profit!'
10 | ];
11 |
12 | // don't change the Component name
13 | "App"
14 | export default function App() {
15 |   return (
16 |     <ul>
17 |       {DUMMY_TODOS.map(todo =>
18 |         <Todo text={todo} />)
19 |     );
20 |   }
21 | }
```

Components

```
</TabButton>
<TabButton onSelect={() => handleSelect('jsx')}>JSX</Ta
<TabButton onSelect={() => handleSelect('props')}>Props<
<TabButton onSelect={() => handleSelect('state')}>State<
  ...
```

App.jsx

```
1 | export default function TabButton({ children, onSelect, isSelected }) {
2 |   console.log('TABBUTTON COMPONENT EXECUTING');
3 |   return (
4 |     <li>
5 |       <button className={isSelected ? 'active' : undefined} onClick={onSel
6 |         <children>
7 |       </button>
8 |     </li>
9 |   );
10 | }
```



React adv

29 April 2024 12:29

General syntax : where a div is used to wrap all the sibling, but this adds an extra div, which can be avoided using this two methods:

1. Old syntax: fragment is used, which needs to be imported from react.
2. Here only <> is used. No import needed(Modern syntax)



Instead of having all the components in the main app component, we should make individual file every component.

Final app component:

```
import Header from './components/Header/Header.jsx';
import CoreConcepts from './components/CoreConcepts.jsx';
import Examples from './components/Examples.jsx';

function App() {
  return (
    <>
      <Header />
      <main>
        <CoreConcepts />
        <Examples />
      </main>
    </>
  );
}

export default App;
```

Component:

```
import CoreConcept from './CoreConcept.jsx';
import { CORE_CONCEPTS } from './data.js';

export default function CoreConcepts() {
  return (
    <section id="core-concepts">
      <h2>Core Concepts</h2>
      <ul>
        [<CoreConcept key={conceptItem.title} {...conceptItem} />]
      </ul>
    </section>
  );
}
```



```
import { useState } from 'react';
import TabButton from './TabButton.jsx';
import { EXAMPLES } from './data.js';

export default function Examples() {
  const [selectedTopic, setSelectedTopic] = useState();
  function handleSelect(selectedButton) {
    // selectedButton => {components, 'jsx', 'props', 'state'}
    setSelectedTopic(selectedButton);
    // console.log(selectedTopic);
  }
  let tabContent = <p>Please select a topic.</p>;

```

Here the section element is used several

Time, so creating a component for section:

```
Examples.jsx ⑥ Section.jsx U ●
1 export default function Section({ title, children }) {
2   return (
3     <section>
4       <h2>{title}</h2>
5       {children}
6     </section>
7   );
8 }
```

Import that in App.jsx

```
import Section from './Section.jsx';
import { EXAMPLES } from './data.js';
```

Modify the section element into the section component :

```
return (
  <Section title="Examples" id="examples">
    <menu>
      <TabButton
        isSelected={selectedTopic === 'components'}
        onSelect={() => handleSelect('components')}
      >
        Components
      </TabButton>
      <TabButton
        isSelected={selectedTopic === 'jsx'}
        onClick={() => handleSelect('jsx')}
      >
        JSSX
      </TabButton>
    </menu>
  </Section>
);
```

Forwarding Props:

Now to get all the classes in the component in that we need to use the prop:

```
Examples.jsx M ⑥ Section.jsx U ● ⑥ TabButton.jsx ...
1 export default function Section({ title, children, ...props }) {
2   return (
3     <section {...props}>
4       <h2>{title}</h2>
5       {children}
6     </section>
7   );
8 }
```

Using this for any number of classes, ID etc. will be called.

After updating:

```
Examples.jsx M ⑥ Section.jsx U ⑥ TabButton.jsx ...
1 export default function TabButton({ children, isSelected, ...props }) {
2   console.log('TABBUTTON COMPONENT EXECUTING');
3   return (
4     <li>
5       <button className={isSelected ? 'active' : undefined} {...props}>
6         {children}
7       </button>
8     </li>
9   );
10 }
11 <menu>
12   <TabButton
13     isSelected={selectedTopic === 'components'}
14     onClick={() => handleSelect('components')}
15   >
16     Components
17   </TabButton>
18   <TabButton
19     isSelected={selectedTopic === 'jsx'}
20     onClick={() => handleSelect('jsx')}
21   >
22     JSSX
23   </TabButton>
24 </menu>
25 
```

```

    >
    Components
  </TabButton>
<TabButton>

```

```

    >
    isSelected={selectedTopic === 'components'}
    onClick={() => handleSelect('components')}
  >
    Components

```

replace for all of them.

Same Forwarding props can be used in TabButton, so we don't have to use the onSelect.

```

  Examples.jsx M  Section.jsx U  TabButton.jsx ...
1  export default function TabButton({ children, onSelect, isSelected }) {
2    console.log('TABBUTTON COMPONENT EXECUTING');
3    return (
4      <li>
5        <button className={isSelected ? 'active' : undefined} onClick={onSelect}
6          | {children}
7        </button>
8      </li>
9    );
10  }
11
12
13

```

components

- > Header
- Tabs.jsx**
- CoreConcept.jsx
- CoreConcepts.jsx
- Examples.jsx
- Section.jsx
- TabButton.jsx
- App.jsx
- JS data.js

JSX Slots

```

  Examples.jsx M  Tabs.jsx U  Section.jsx  TabButton.jsx ...
1  export default function Tabs({ children, buttons }) {
2    return (
3      <>
4        <menu>{buttons}</menu>
5        {children}
6      </>
7    );
8  }

```

Here we are making a common component tabs.jsx which can be reused

```

  Examples.jsx  Tabs.jsx U  Section.jsx  TabButton.jsx ...
30
31  return (
32    <Section title="Examples" id="examples">
33      <Tabs
34        buttons={
35          <>
36            <TabButton
37              isSelected={selectedTopic === 'components'}
38              onClick={() => handleSelect('components')}
39            >
40              Components
41            </TabButton>
42            <TabButton
43              isSelected={selectedTopic === 'jsx'}
44              onClick={() => handleSelect('jsx')}
45            >
46              JSX

```

```

    >
    Props
  </TabButton>
  <TabButton
    isSelected={selectedTopic === 'state'}
    onClick={() => handleSelect('state')}
  >
    State
  </TabButton>
  </>
  <tabContent>
  </Tabs>
  <menu></menu>
</Section>

```

```

  Examples.jsx M  Tabs.jsx U  Section.jsx  TabButton.jsx ...
54
55  <TabButton
56    isSelected={selectedTopic === 'state'}
57    onClick={() => handleSelect('state')}
58  >
59    State
60  </TabButton>

```

Using a extra prop to customize the component:

```

buttonContainer
  Examples.jsx M  Tabs.jsx M  Section.jsx  TabButton.jsx ...
31
32  return (
33    <Section title="Examples" id="examples">
34      <Tabs
35        buttonsContainer={Section}
36        buttons={[
37          <>
38            <TabButton
39              isSelected={selectedTopic === 'components'}
40              onClick={() => handleSelect('components')}
41            >
42              Components
43            </TabButton>
44            <TabButton
45              isSelected={selectedTopic === 'jsx'}
46              onClick={() => handleSelect('jsx')}

```

```
41 |     Components
42 |     </TabButton>
43 |     <TabButton
44 |       isSelected={selectedTopic === 'jsx'}
45 |       onClick={() => handleSelect('jsx')}
46 |     >
```

```
55 |       isSelected={selectedTopic === state}
56 |       onClick={() => handleSelect('state')}
57 |     >
58 |     State
59 |     </TabButton>
60 |   </>
61 | }
62 | >
63 | {tabContent}
64 | </Tabs>
65 | </Section>
66 | ];
67 | }
```

```
1 | export default function Tabs({ children, buttons, buttonsContainer }) {
2 |   return (
3 |     <>
4 |       <menu>{buttons}</menu>
5 |       {children}
6 |     </>
7 |   );
8 | }
```

```
1 | export default function Tabs({ children, buttons, buttonsContainer }) {
2 |   const ButtonsContainer = buttonsContainer;
3 |   return (
4 |     <>
5 |       <ButtonsContainer>{buttons}</ButtonsContainer>
6 |       {children}
7 |     </>
8 |   );
9 | }
```

Default props values:

```
1 | default function Tabs({ children, buttons, ButtonsContainer = 'menu' }) {
2 |   const ButtonsContainer = buttonsContainer;
3 |   <div>
4 |
5 |     <ButtonsContainer>{buttons}</ButtonsContainer>
6 |     {children}
7 |   </div>
8 |
9 |
10| }
```

```
<Section title="Examples" id="examples">
  <Tabs
    buttonsContainer="div"
    buttons={
      <>
```

The buttonContainer dynamically

If we want to call another component in the buttonContainer the we need to use {} and If we are using div or menu something like that then we have to use "".

Here in the Tabs.jsx component the buttonContainer wont get reconnised as a component, because the system will try to fina a default component with the same name because it is starting with small letter

There are two solution for it:

1. Make a variable which start from the capital letter and assign that variable to that component, then call that variable as the component
2. Or else start the component name with capital letter.

New note:

```
App.jsx  X  Players.jsx
src > App.jsx > App
1 import Player from "./components/Players"
2 function App() {
3
4   return (
5     <main>
6       <div id="game-container">
7         <ol id="players">
8           <Player initialName="player1" symbol="X" />
9           <Player initialName="player2" symbol="O" />
10        </ol>
11        GAME BOARD
12      </div>
13      LOG
14    </main>
15  );
16}
17
18 export default App
19
```

#calling the component player in the main app component.

App.jsx

Players.jsx X

```
src > components > ⚘ Players.jsx > ⚘ Player > ⚘ handleClick
1 import { useState } from "react";
2
3 export default function Player({initialName, symbol}){
4     const [playerName, setplayerName] = useState(initialName);
5     const [isEditing, setIsEditing] = useState(false);
6     function handleClick(){
7         setIsEditing((editing) => !isEditing);
8     }
9     function handleChangeName(event){
10         setplayerName(event.target.value);
11     }
12     let player = <span className="player-name">{playerName}</span>;
13     let clicked="edit";
14     if(isEditing) {
15         clicked = "save";
16         player= <input type="text" required value={playerName} onChange={handleChangeName} />;
17     }
18     return(
19     <li>
20     <span className="player">
21     {player}
22     <span className="player-symbol">{symbol}</span>
23     </span>
24     <button onClick={handleClick} >{clicked}</button>
25     </li>
26     );
27 }
```