



Licenciatura  
Multimédia

# Programação III

Frederico Fonseca

[ffonseca@ismt.pt](mailto:ffonseca@ismt.pt)



Imagem retirada do website  
<https://unsplash.com/photos/MuJHwDHbXUk>

# Sumário

- Props e States
- React Hooks
- JSX
- Mapas e chaves

Texto de apoio:

Morgan, J. (2021). **How To Code in React.js** [E-book].

Consultado em <https://www.digitalocean.com/community/books/how-to-code-in-react-js-ebook>

# Props e State

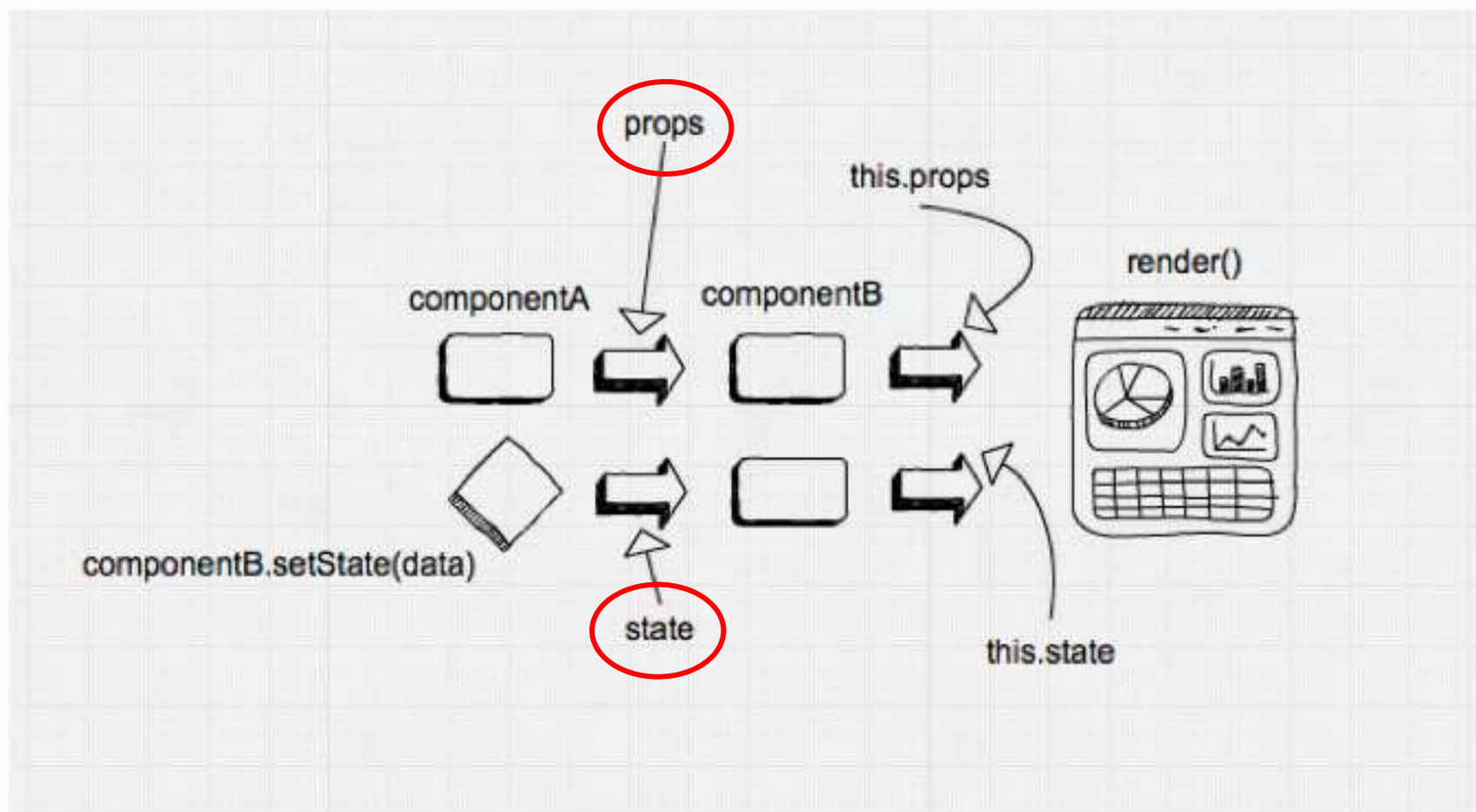


Figura 1 – Ilustração de Props vs States em React

# Props e State

- As componentes lidam com a lógica de transformação dos dados, incluindo a apresentação dos mesmos;
  - **Recebem, processam e retornam dados;**
- Para tal é necessário utilizar **Props** ou **State**;
  - Estes determinam o que o componente processa e como ele se comporta internamente;
- Independentemente do tipo de componente, o nome deste deverá ser definido em maiúsculas - caso contrário dará erro!!!

# Props

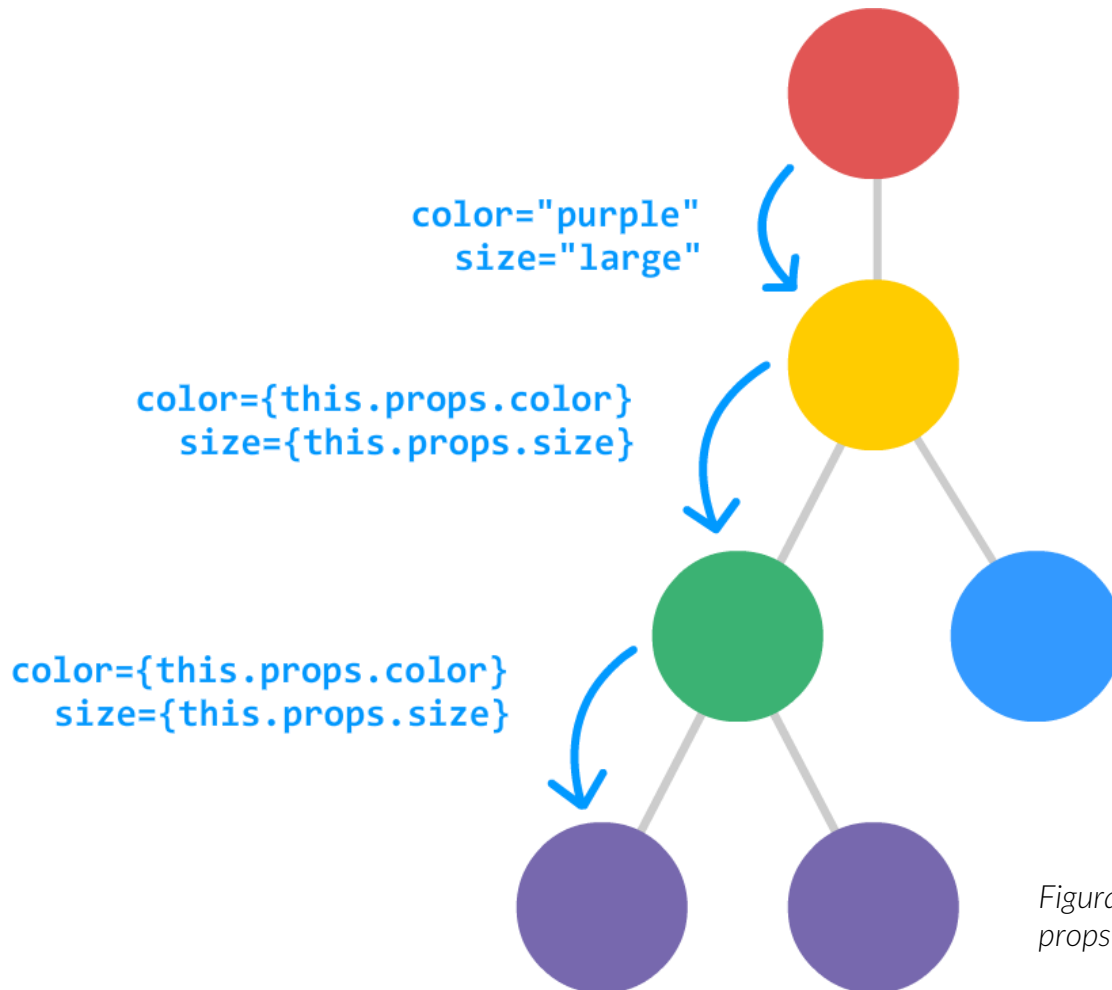


Figura 2 – Ilustração da passagem de props entre componentes

# Props

- O **Props** recebe os dados do componente pai (*parent*) em modo **apenas de leitura** (*read only*);
  - Significa que um componente pai pode transmitir dados para os seus filhos (via *Props*), mas os componentes filho não o podem modificar;
- O *Props* deve ser imutável e executável de cima para baixo;

# Props

- Se os componentes forem do tipo funcional – i.e., funções JS, o **Props** é o parâmetro da função;
  - Um componente aceita um parâmetro (*Props*), processa a informação e renderiza em código JSX;



# State

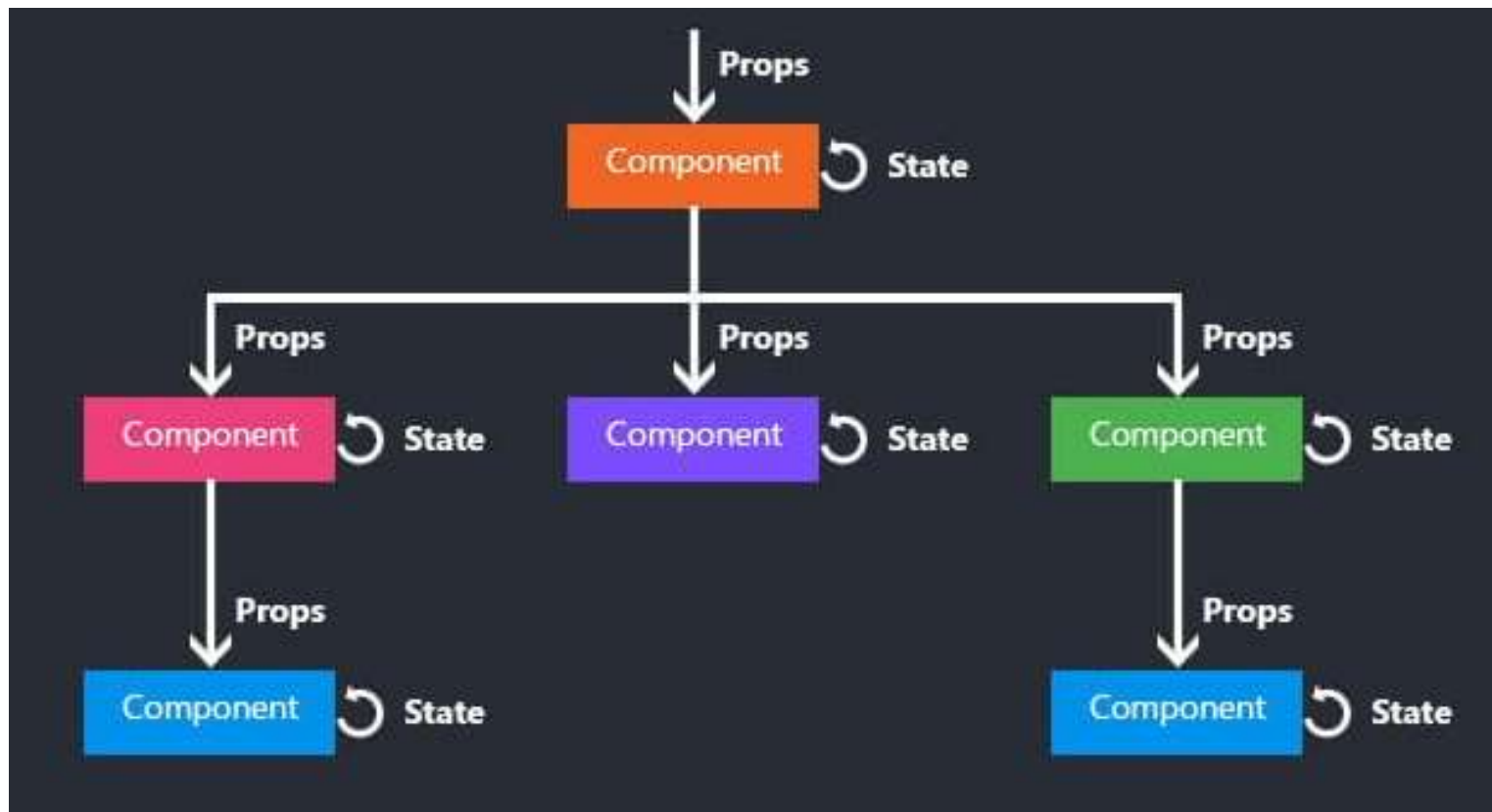


Figura 3 – States em componentes React



- É uma propriedade que pertence ao componente onde é declarado - o seu âmbito está limitado ao componente atual;
- Um componente pode iniciar o seu estado (*state*) e atualizá-lo sempre que necessário;
  - Quando o estado (*state*) vai para além do âmbito atual, passa a ser tratado como *Props*;

# State

- O estado (*state*) é usado para comunicação dentro do próprio componente;



# Exemplo

```
const Person = (props) => {  
  return (  
    <h1>  
      Name: {props.name}, Age: {props.age}  
    </h1>  
  )  
}
```

Figura 3a – Exemplo de um componente Person que recebe os props sem estrutura

```
const Person = ({ name, age }) => {  
  return (  
    <h1>  
      Name: {name} Age: {age}  
    </h1>  
  )  
}
```

Figura 3b – Exemplo de um componente Person que recebe os props estruturados

```
<Person name="João Pedro" age={29} />
```

# Exemplo



```
const Person = ({ name = 'Rui Silva', age = '18' }) => {  
  return (  
    <h1>  
      Name: {name}, Age: {age}  
    </h1>  
  )  
}
```

Figura 3c – Exemplo de Props estruturados com valor por defeito



```
const Person = ({ name, age }) => {  
  return (  
    <h1>  
      Name: {name}, Age: {age}  
    </h1>  
  )  
}  
  
Person.defaultProps = {  
  name: 'Rui Silva',  
  age: '18',  
}
```

Figura 3c – Exemplo de Props estruturados com valor por defeito

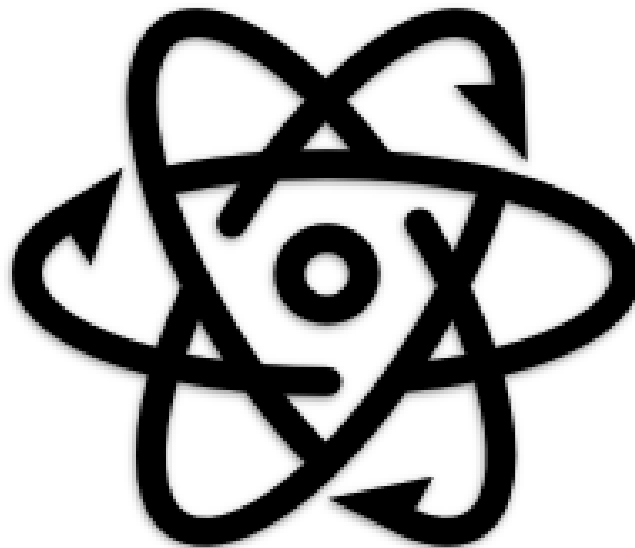
# Exemplo

```
const Component = ({ children }) => {  
  return <div>{children}</div>  
}
```

Figura 3d – Exemplo de um componente que recebe um outro componente como parâmetro

```
const Exemplo = () => {  
  return (  
    <Component>  
      <h1>Hello World</h1>  
    </Component>  
  )  
}
```

# React Hooks



# React Hooks

- Permite utilizar **estados** (*states*) em componentes que não são do tipo *classe* – as chamadas componentes funcionais;
- Instância um estado (*state*) recorrendo a *hooks*;
  - O React irá preservar o estado entre renderizações;
- Continua válida a regra que o estado deve ser considerado **imutável**, isto é, não deve ser alterado de forma direta;
  - Por exemplo, num *state* do tipo array, não é permitido utilizar o *push()* para adicionar valores;

# React Hooks

- Resumidamente, os **hooks** são funções JS que permitem utilizar recursos (states e *lifecycle*) em componentes funcionais;
  - Representam um tipo especial de lógica reutilizável nas interfaces gráficas (UI) com restrições na forma como podem ser invocados;
- Todos os hooks necessitam de ser importados no início do componente;

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





# React Hooks

- Ao utilizarmos hooks somos obrigados a cumprir **três regras** muito importantes:
  - Utilizar *hooks* apenas em componentes funcionais;
  - Não invocar os *hooks* dentro de *loops*, condições, funções embutidas ou estruturas try/catch/finally;
  - O hook deve ser obrigatoriamente invocado na definição (parte superior) do componente;

# React Hooks

- Exemplo de um *hook* básico:

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

function Exemplo() {
  const [idade, setIdade] = useState(20);

  return (
    <div>
      <p>Idade atual: {idade}</p>
      <button onClick={() => setIdade(idade + 1)}>
        Feliz Aniversário!
      </button>
    </div>
  );
}
```

Idade atual 20  
Feliz Aniversário!

# React Hooks

- Exemplo de declaração de vários *hooks*:

```
// Exemplo de hooks do tipo string e array
const [query, setQuery] = useState("")
const [listaPessoas, setListaPessoas] = useState([])
```

**Nota:** no segundo exemplo o state é acessível via `listaPessoas` e mutável via `setListaPessoas` - este método também é chamado de *reducer*.

# React Hooks

- Para alterar o valor de um *state* necessitamos de recorrer obrigatoriamente ao *reducer*;
- No caso de um array, numa situação normal, utilizamos o método `push()` para adicionar um valor;
- No entanto, quando recorremos a *hooks*, necessitamos de utilizar o `concat()`;
- Esta situação ocorre pelo facto do *React* estar à espera de receber o array no seu estado final (depois de alterado), o que acontece com o `concat()`, mas não com o `push()`, que retorna o tamanho (length) do array;

# React Hooks

```
// não utiliza um 'wrapper function' (não recomendado)
setListaPessoas(listaPessoas.concat("xpto"))

// utiliza um 'wrapper function' (recomendado)
setListaPessoas(listaPessoas => listaPessoas.concat("xpto"))

// utiliza o operador 'spread', mas sem 'wrapper function' (não recomendado)
setListaPessoas([...listaPessoas, "xpto"])

// utilizando o operador 'spread' com 'wrapper function' (recomendado)
setListaPessoas(listaPessoas => [...listaPessoas, "xpto"])
```

- Para mais informações ler o artigo disponível [aqui](#).

# useEffect

- O `useEffect` é um hook que permite aceder aos eventos do ciclo de vida (*lifecycle*) de um componente funcional;
  - Os métodos são: `mount`, `update` e `unmount`;

```
useEffect(() => {  
  console.log('mounts')  
  
  return () => {  
    console.log('unmounts')  
  }  
}, []);
```

```
useEffect(() => {  
  document.title = `Carregou no botão ${count} vezes`  
}, [count]);
```

# useEffect

```
const Exemplo = () => {  
  const [count, setCount] = useState(0);  
  
  useEffect(() => {  
    console.log('mounts')  
  
    return () => {  
      console.log('unmounts')  
    }  
  }, []);  
  
  useEffect(() => {  
    document.title = `Carregou no botão ${count} vezes`  
  }, [count]);  
  
  return (  
    <div>  
      <button onClick={() => setCount(count + 1)}>Clique aqui</button>  
    </div>  
  )  
}
```

Figura 4 – Exemplo de um componente em JSX que faz uso do hook `useState` e `useEffect` para gestão de estados e eventos do ciclo de vida do componente

# useContext

- O `useContext` é um hook que retorna dados num determinado contexto;



```
const ThemeContext = createContext(null);
```

Figura 5a – Instancia de criação de um contexto (ThemeContext) utilizando o hook `useContext`



```
<ThemeContext.Provider value="light">  
  <Exemplo />  
</ThemeContext.Provider>
```

Figura 5b – Definição do fornecedor do contexto (ThemeContext) e do seu respetivo valor (neste caso, a expressão "light")



```
const Exemplo = () => {  
  const tema = useContext(ThemeContext)  
  return (  
    <div>  
      <p>O tema atual é o: {tema}</p>  
    </div>  
  )  
}
```

Figura 5c – Referência a um contexto específico (ThemeContext) e sua utilização



# useContext

```
const ThemeContext = createContext(null);

function App() {
  return (
    <ThemeContext.Provider value="light">
      <Exemplo />
    </ThemeContext.Provider>
  )
}

const Exemplo = () => {
  const tema = useContext(ThemeContext) // retorna 'light'
  return (
    <div>
      <p>O tema atual é o: {tema}</p>
    </div>
  )
}
```

Figura 6 – Exemplo de uma app que faz uso do hook useContext para passar valores no contexto pretendido

# useMemo

- O `useMemo` é um hook que permite armazenar em cache o resultado de um cálculo entre renderizações;
  - É o chamado hook de otimização de desempenho;



```
const memoIncrement = useMemo(() => {  
  return () => setCount((c) => c + 1)  
}, []);
```

# useMemo

```
const Exemplo = () => {  
  const [count, setCount] = useState(0);  
  
  const memoIncrement = useMemo(() => {  
    return () => setCount((c) => c + 1)  
  }, []);  
  
  return (  
    <div>  
      <p>{count}</p>  
      <button onClick={memoIncrement}>+</button>  
    </div>  
  )  
}
```

Figura 8 – Componente Exemplo que recorre ao hook `useMemo` para armazenar o último valor retornado pelo `setCount()`

- O `useRef` é um hook que cria um **objeto** mutável que contém uma **única propriedade** – de nome `current`;
  - O objeto terá como identificador a palavra `ref`;
  - É necessário definir um valor inicial (`initialValue`) e que pode ser `null`, 0, etc.;
- O objeto retornado persistirá ao longo de todo o tempo de vida do componente, a menos que seja alterado manualmente;

```
const Exemplo = () => {  
  const inputRef = useRef(null);  
  
  const handleClick = () => {  
    inputRef.current.focus();  
  }  
  
  return (  
    <div>  
      <input ref={inputRef} type="text" />  
      <button onClick={handleClick}>Clique aqui!</button>  
    </div>  
  )  
}
```

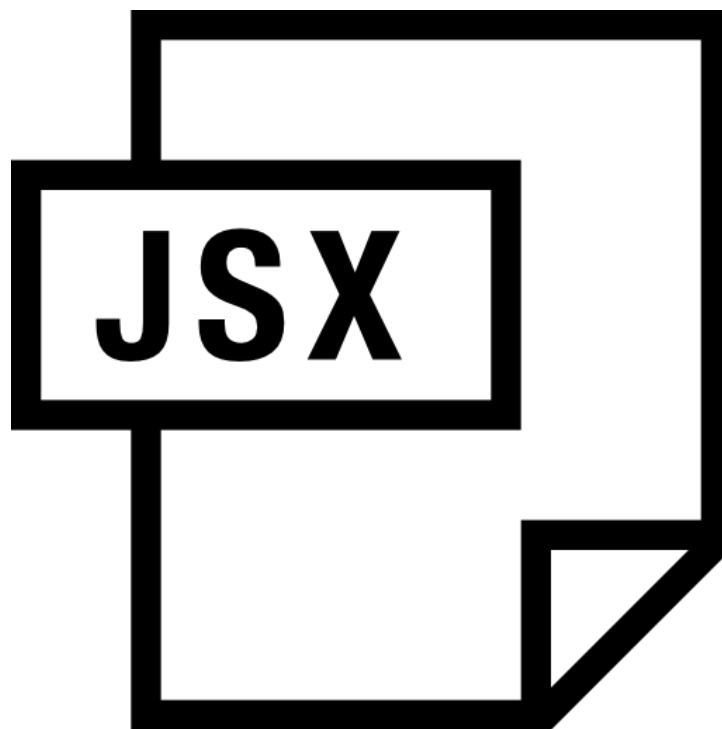
Figura 9 – Componente Exemplo que recorrendo ao hook useRef para armazenar o último valor retornado pelo setCount()

- IMPORTANTE:
  - Pode armazenar informações entre renderizações - ao contrário das variáveis que são redefinidas (iniciadas) a cada renderização;
  - A alteração do seu valor não desencadeia uma nova renderização - ao contrário das variáveis de estado que desencadeiam uma nova renderização;
  - A informação é local para cada componente - ao contrário das variáveis “externas”, que são partilhadas;

- IMPORTANTE:
  - A alteração de uma `ref` não desencadeia uma nova renderização - pelo que as `refs` não são adequadas para armazenar informações que pretendemos apresentar no ecrã (para tal devemos utilizar o `useState`);



Licenciatura  
Multimédia





- JSX é um formato **JavaScript XML** usado em aplicações React, embora não exclusivo, com o objetivo de tornar mais fácil a criação de aplicações em React;
  - Torna o código mais legível, confiável e fácil de modificar;
  - O JSX usa a sintaxe HTML/XML para criar elementos e componentes;
  - O compilador de JSX converte o código em JS puro, chamado *Vanilla JS*, sendo assim interpretado pelos *browsers*;

- O JSX permite escrever elementos HTML em JavaScript e colocá-los no DOM sem recorrer aos métodos `createElement()` e/ou `appendChild()`;
- O JSX não é interpretado pelo *browser*, o React usa o **Babel** para interpretar e transformar em código vanilla JS;

```
<h1 className="title">Hello World</h1>
```



```
React.createElement(  
  "h1",  
  { className: "title" },  
  "Hello World"  
);
```

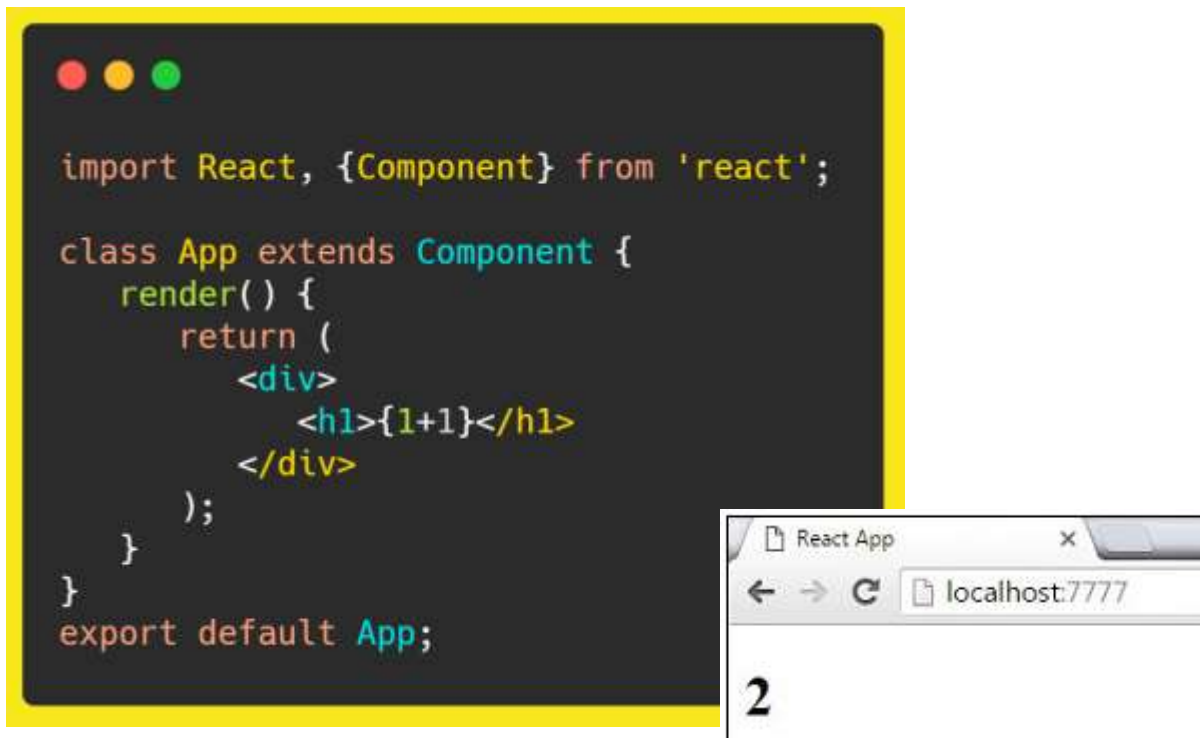
Figura 10 – Exemplificação do código HTML vs vanilla JS

```
const nav = (  
  <nav className="menu">  
    <ul>  
      <li><a href="#">Home</a></li>  
      <li><a href="#">About</a></li>  
    </ul>  
  </nav>  
)
```



```
var nav = React.createElement(  
  "nav",  
  { className: "menu" },  
  React.createElement(  
    "ul",  
    null,  
    React.createElement(  
      "li",  
      null,  
      React.createElement(  
        "a",  
        { href: "#" },  
        "Home"  
      )  
    ),  
    React.createElement(  
      "li",  
      null,  
      React.createElement(  
        "a",  
        { href: "#" },  
        "About"  
      )  
    )  
  )  
);
```

- Expressões JavaScript podem ser usadas dentro do JSX;
- É necessário colocar o código entre chavetas ({ });



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

class App extends Component {
  render() {
    return (
      <div>
        <h1>{1+1}</h1>
      </div>
    );
  }
}

export default App;
```

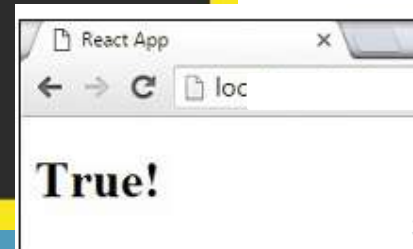
The image shows a code editor window with the above code. To the right, a web browser window titled 'React App' is open, displaying the rendered output of the code: a heading '2' (the result of 1+1) inside a container. The browser's address bar shows 'localhost:7777'.

- Não podemos usar instruções if.. else dentro do JSX, mas podemos usar expressões condicionais;

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

class App extends Component {
  render() {
    var i = 1;
    return (
      <div>
        <h1>{i == 1 ? 'True!' : 'False'}</h1>
      </div>
    );
  }
}

export default App;
```



- O React recomenda o uso de **estilos embebidos** (*inline*);
  - Quando necessário, deve ser usada sintaxe *CamelCase*;
- O React acrescenta automaticamente 'px' (pixels) depois do valor numérico, em elementos específicos;

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

class App extends Component {
  render() {
    var myStyle = {
      fontSize: 100,
      color: '#FF0000'
    }
    return (
      <div>
        <h1 style={myStyle}>Header</h1>
      </div>
    );
  }
}

export default App;
```



- O JSX foi criado para ser parecido com HTML, mas com o poder de criação e utilização de componentes reutilizáveis;
- Ao retornar JSX de uma função, podemos criar essa reutilização:

```
function User(props) {  
  return <div>User: {props.name} com {props.age}</div>  
}  
  
function App() {  
  return (  
    <div>  
      <User name="João" age="34" />  
      <User name="Abel" age="25" />  
    </div>  
  );  
}
```

- Diferentes tipos de valores que podem ser enviados:

```
<ExemploComponente
  nome="Pedro"      // String
  idade={34}        // Number
  ativo={true}      // Boolean
  hobbies=[         // Array
    Jogar computador,
    'Longas caminhadas na praia'
  ]
  localizacao={{    // Objeto
    cidade: 'Coimbra',
    pais: 'PT'
  }}
  onRemove={apagaUtilizador} // referência para uma função
  onSave={() => { // uma função inline
    console.log('Olá')
  }}
/>
```



# JSX className

- **className** - substitui o atributo **class** no HTML, pois **class** é uma palavra reservada em JS;



```
// JSX
```

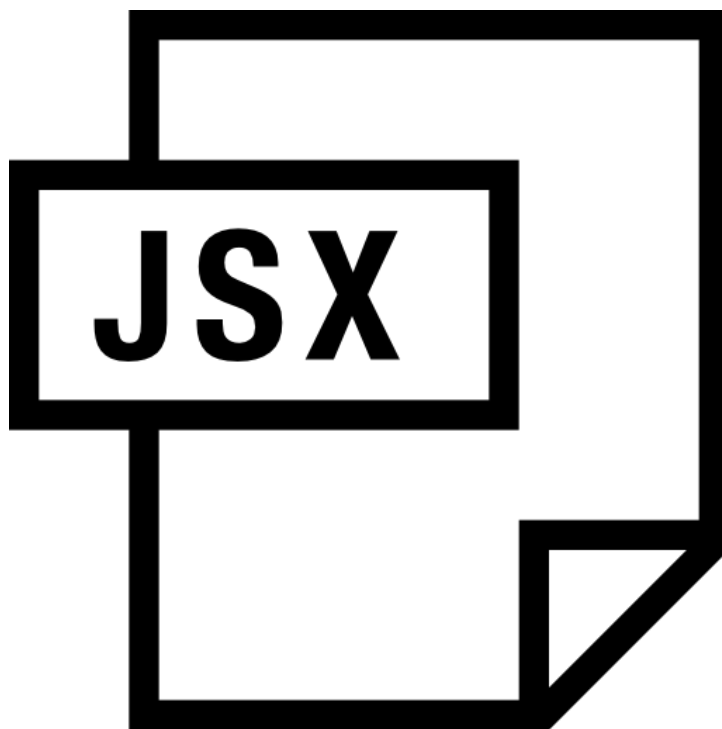
```
<div className="active">Hello</div>
```

```
// HTML
```

```
<div class="active">Hello</div>
```



Licenciatura  
Multimédia



# JSX Mapas e Chaves

- Tendo em conta o seguinte exemplo:
  - Os dois elementos `<li>` vão ser filhos (*Props children*) para o elemento `<ul>`;

```
function UserList() {  
  const users = ['João', 'Abel'];  
  return (  
    <ul>  
      <li>{users[0]}</li>  
      <li>{users[1]}</li>  
    </ul>  
  );  
}
```

# JSX Mapas e Chaves

- O código anterior não é dinâmico e por cada novo utilizador (*user*), teríamos que adicionar manualmente no código;
- A solução iterativa consiste em passar por **props** o *array* e utilizar a função `map()` para iterar os valores;

```
function UserList(props) {  
  return (  
    <ul>{props.users.map(name => {  
      return <li>{name}</li>  
    })}  
    </ul>  
  );  
}  
<UserList users={['Adolfo', 'Raimundo']} />
```

# JSX Mapas e Chaves

- É necessário utilizar **chavetas** no JSX para passar um valor;
- Não é possível utilizar o *while*, *if*, *switch*, *for* ou *foreach* no JSX, pois estes métodos não retornam valores;
- O React faz o mapeamento dos componentes que constrói e em que parte da DOM eles estão associados;
- O React faz esse processo de forma automática e transparente, exceto no caso de fornecermos ao JSX um array (quando utilizamos *map*);
- É necessário fornecer uma chave única para que o React possa controlar os elementos que constrói;

# JSX Mapa e Chaves

- A função `map()` fornece o índice (*index*) que pode ser utilizado como chave no segundo argumento da função:

```
{props.users.map((name, index) => {  
  return <li key={index}>{name}</li>  
})}
```

# JSX Mapas e Chaves

- Informações importantes sobre as chaves:
  - A **key** necessita de ser única dentro do array, não para toda a aplicação ou componentes;
  - As **keys** podem ser do tipo *string*, desde que sejam únicas;
  - Usar as **keys** como sendo os índices dos arrays, pode ser uma má ideia;
  - O ideal é utilizar as **keys** como os ID's das BD, pois mesmo que um registo seja eliminado o índice (*index*) do array muda, mas o da tabela não;