REACT.

MATERIAL DE APOIO

descomplica

React

Fundamentos do react

Professor: Ayrton Teshima

Tutor: Rodrigo Vasconcelos

Introdução

React não é um framework. Há muitos debates sobre react. É um framework ou uma biblioteca? React não é exatamente um framework. A estrutura é uma solução completa. Tudo o que você precisa para fazer seu aplicativo está embutido. Um framework geralmente quer que você codifique tudo de uma certa maneira. Se você tentar adiar dessa maneira, a estrutura geralmente acaba brigando com você sobre isso. Porém no react você não tem tudo para construir sua aplicação.

Objetivos da aula

- Compreender mais dos fundamentos do React
- Criar seus primeiros componentes React
- Entender sua composição básica

Resumo

Existem duas formas de criar um componente. A mais tradicional e correta é criar através de funções.

Funções

Para desenvolvermos componentes React através de funções, você deve definir uma função que retorna um elemento React.

No exemplo a seguir, criamos esse elemento através da função do React *createElement.* Logo depois, utilizamos o método *render* do ReactDOM para renderizar o componente que acabamos de criar dentro de uma tag de id app.

```
function Hello() {
  return (
    React.createElement('h1', null, 'Olá mundo')
  )
}

ReactDOM.render(
  React.createElement(Hello),
  document.querySelector('#app')
)
```

Fonte: autoral.

O primeiro argumento de *createElement* é o nome da tag que queremos utilizar, o segundo são seus atributos e o terceiro é seu elemento filho (que nesse caso é um texto apenas). Vamos conhecer mais sobre criação de elementos React mais a frente.

Classes

Também podemos criar componentes como classe, que você vai encontrar ainda em alguns artigos, documentações e projetos legados.

Toda classe que é um componente React deve ter o método *render* que retorna um elemento React e deve extender de *React.Component*

```
class Hello extends React.Component {
  render() {
    return (
      React.createElement('h1', null, 'Olá mundo')
    )
  }
}
```

Fonte: autoral.

Criando HTML com React

Tags HTML

No exemplo anterior, vimos o componente *Hello* que renderiza a tag *h1*. Agora, não estamos criando um componente através de função ou classe, estamos gerando o elemento *span* e já o renderizando imediatamente.

Toda vez que queremos utilizar uma tag HTML, precisamos passar o nome da tag como string no primeiro parâmetro do método *createElement*.

No exemplo a seguir, estamos criando a tag *span*.

```
ReactDOM.render(
  React.createElement('span', null, 'Olá'),
  document.querySelector('#app')
)
```

Fonte: autoral.

Atributos

Introdução à atributos

Tags HTML permitem passar atributos para que você possa customizar aquela tag. Exemplos:

A tag <*a*> permite passar o *href* para que você especifique a URL que vai ser acessada ao clicar no link

A tag *img* aceita o atributo *src* para que você indique a imagem a ser carregada

No React não é diferente, você pode passar atributos para as tags HTML e também para seu próprio componente.

Vamos voltar ao exemplo do componente *Hello.*

```
function Hello() {
  return (
    React.createElement('h1', null, 'Olá mundo')
  )
}

ReactDOM.render(
  React.createElement(Hello),
  document.querySelector('#app')
)
```

Fonte: autoral.

O componente *Hello* não tem atributos e também ainda não permite receber. Como já falamos, os atributos são definidos no segundo parâmetro do método *createElement*.

No segundo parâmetro você passa um objeto com todos os atributos que deseja definir, onde a chave de cada propriedade é o nome do atributo e seu valor, é o valor do atributo.

Vamos definir um *id* para nosso span:

```
function Hello() {
  return (
    React.createElement('h1', { id: 'titulo' }, 'Olá mundo')
  )
}
```

Fonte: autoral.

Veja como é simples. Os nomes dos atributos que você passa possuem o mesmo nome real dos atributos que aquela tag HTML espera.

Existem pouquíssimas exceções para isso. Um exemplo muito comum de atributo que tem nome diferente quando utilizamos React é o *class.*

Se você quiser adicionar uma classe para sua tag no React, você utiliza a propriedade *className* no lugar de *class*.

```
function Hello() {
  return (
    React.createElement('h1', { id: 'titulo', className: 'titulo-1' }, 'Olá
mundo')
  )
}
```

Fonte: autoral.

Recebendo atributos

Seu componente pode ter atributos próprios para que seja customizável. Vamos fazer com que nosso componente *Hello* aceite um título e um id.

```
function Hello(props) {
  return (
    React.createElement('h1', { id: props.id, className: 'titulo-1' },
props.label)
  )
}
```

Fonte: autoral.

Como podemos ver na imagem acima, os atributos que o componente *Hello* espera chegam através de um parâmetro que chamamos de *props (propriedades)*.
Nesse objeto, vamos receber as propriedades *id* e *label.* Veja que utilizamos essas propriedades na criação do elemento *h1*.

E como passamos esses props? Simples. Do mesmo jeito que fazemos para atributos de uma tag HTML normal. Veja o método *render* do ReactDOM.

```
function Hello(props) {
  return (
    React.createElement('h1', { id: props.id, className: 'titulo-1' },
props.label)
  )
}

ReactDOM.render(
  React.createElement(Hello, { id: 'titulo', label: 'Olá dev!' }),
  document.querySelector('#app')
)
```

Fonte: autoral.

Atributos e componentes de classes

Se você estiver com um componente do tipo classe, você acessa as *props* passadas para seu componente através de *this.props*.

```
class Hello extends React.Component {
  render() {
    return (
      React.createElement('h1', { id: this.props.id, className: 'titulo-1'
}, this.props.label)
    )
  }
}

ReactDOM.render(
  React.createElement(Hello, { id: 'titulo', label: 'Olá dev!' }),
  document.querySelector('#app')
)
```

Fonte: autoral.

Caso você tenha em seu componente do tipo classe o *constructor* definido, você precisa invocar *super* passando o *props* para que seu componente tenha acesso a ele.

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

  render() {
    return (
      React.createElement('h1', { id: this.props.id, className: 'titulo-1'
}, this.props.label)
    )
  }
}

ReactDOM.render(
  React.createElement(Hello, { id: 'titulo', label: 'Olá dev!' }),
  document.querySelector('#app')
)
```

Fonte: autoral.

Create React App

O que é o create-react-app

O create-react-app é um toolchain que serve para criar toda uma estrutura de projeto pré-configurada para que possamos focar apenas em desenvolver.

Vamos começar a explorar o famoso JSX, que é a forma como você codifica sua interface com React ao invés de utilizar React.createElement. Porém, o navegador não conhece JSX. Por isso precisamos passar por transpilação, momento em que o seu JSX será transformado em código JavaScript de verdade.

Podemos fazer essa configuração na mão, mas como disse, o create-react-app vem para ajudar nessa etapa.

Pré-requisitos e instalação

O create-react-app é uma ferramenta de linha de comando muito fácil de instalar, em qualquer sistema operacional utilizado. Você só precisa ter a versão do Node a partir da 14 em seu sistema operacional. Dê preferência para instalar a última versão.

Quando você instala o Node, você vai ganhar o npm e npx.

Com o npx você consegue executar um pacote sem precisar baixar em sua máquina, e vamos justamente executar o create-react-app através dele para gerar um projeto

Execute em seu terminal:

npx create-react-app react-estudos

Com esse comando você estará criando dentro da pasta react-estudos toda a estrutura de arquivos e pastas do seu projeto React. Ao executar o comando, ele fará algumas perguntas sobre preferências de ferramenta, basta selecionar a melhor para você e continuar.

Comandos

Ao finalizar a criação do projeto, acesse a pasta via terminal recém-criada:

cd react-estudos

A seguir, execute o comando para rodar o projeto:

npm start

Aguarde o comando executar, com isso seu projeto estará rodando na porta *3000*. Acesse pelo navegador o caminho: http://localhost:3000

Agora veja seu projeto rodando e pronto para ser desenvolvido.

Run time vs Build time

React é muito conhecido pelo JSX. Porém, JSX não funciona no navegador diretamente. Veja que os componentes gerados pelo create-react-app não utilizam *React.createElement* que fizemos nos exemplos anteriores. Ele utiliza uma sintaxe semelhante ao HTML, para essa sintaxe rodar no navegador, o código precisa antes ser transpilado.

O código que você escreve não é o código que está sendo executado pelo navegador. A etapa que o navegador/motor JavaScript está executando seu código é o *run time* e o processo de transpilação / compilação vai acontecer no *build time*.

Ou seja, o JSX que você escreve não é o código que o navegador (motor JavaScript) vai executar, pois antes todo o JSX que você escreveu foi convertido para *React.createElement*.

descomplica

JSX

Sintaxe das tags

Escrever as tags HTML com JSX é muito parecido com HTML de fato. Quer um *span*, um *p*, uma *div*? Simples:

```
function Hello() {
  return (
    <div>
      <p>Meu texto <span>destacado</span></p>
    </div>
  );
}
```

Fonte: autoral.

Atributos

Para definir atributos é exatamente igual ao HTML, basta passar os atributos dentro da tag de abertura:

```
function Hello() {
  return (
    <div id="container">
      <p>Meu texto <span>destacado</span></p>
       <button type="button" id="btn">Aqui</button>
    </div>
  );
}
```

Fonte: autoral.

Classes

As classes das tags devem utilizar o nome *className* e não *class*, que é o utilizado no HTML de verdade.

A explicação para isso vem do processo de transpilação. A palavra *class* é uma palavra reservada no JS que utilizamos quando queremos criar classe. Como o JSX precisa ser transpilado para JS, a palavra class entra em conflito com o class do JS, por isso no React foi preciso mudar o nome.

O mesmo acontece com o atributo *for* da tag *label*. Precisamos utilizar *htmlFor* no lugar já que *for* é uma palavra reservada do JS.

```
function Hello() {
  return (
    <div id="container">
      <p className="texto">Meu texto <span>destacado</span></p>
       <button type="button" id="btn">Aqui</button>
    </div>
  );
}
```

Fonte: autoral.

Criando componente com JSX

Podemos criar componentes à vontade no React e simplesmente utilizá-los como se fossem tags HTML.

Criamos componentes exatamente da forma como fizemos sem JSX, a diferença está na forma como utilizamos. Para informar ao React que aquela função é um componente React, precisamos utilizar o nome da função como se fossem tags HTML, veja o exemplo do componente *Título* abaixo:

```
function Titulo() {
  return <h1>Olá mundo</h1>
}

function App() {
  return (
    <div id="container">
      <Titulo />
      <p class="texto">Meu texto <span>destacado</span></p>
       <button type="button" id="btn">Aqui</button>
    </div>
  );
}
```

Fonte: autoral.

Valores dinâmicos

Se quisermos renderizar um valor dinâmico provinda de uma variável, por exemplo, utilizamos entre chaves

```
function Titulo() {
  const texto = 'Olá mundo!';
  return <h1>{texto}</h1>
}
```

Fonte: autoral.

No exemplo acima, renderizamos o valor da constante *texto* dentro do elemento *h1*. Como o desejado era imprimir o valor da constante texto e não a palavra *texto*, tivemos que utilizar entre chaves.

Todo JavaScript que queira utilizar dentro do JSX precisa estar entre chaves.

Podemos passar por atributo e renderizar:

```
function Titulo(props) {
  return <h1>{props.texto}</h1>
}

function App() {
  return (
    <div id="container">
      <Titulo texto="Meu titulo" />
      <p>Meu texto <span>destacado</span></p>
       <button type="button" id="btn">Aqui</button>
    </div>
  );
}
```

Fonte: autoral.

E, é claro, podemos passar múltiplos atributos

```
const Titulo = ({ texto, numero, pais }) => (
  <h1>{texto} <span>{numero}</span> - <strong>{pais}</strong></h1>
);

function App() {
  return (
    <div id="container">
      <Titulo texto="Meu titulo" numero={30} pais="Brasil" />
      <p>Meu texto <span>destacado</span></p>
       <button type="button" id="btn">Aqui</button>
    </div>
  );
}
```

Fonte: autoral.

Como você pode ver acima, no atributo *numero* eu utilizei chaves para passar um número. Se passarmos tudo entre aspas, tudo vai ser *string.*

Todo valor que seja diferente de *string*, você precisa passar entre chaves. Seja um *number, boolean, array* ou mesmo *object*.

Valor dinâmico em atributo

Veja no componente *Titulo* abaixo que utilizamos o valor dinâmico *pais* para renderizar o seu valor como *className*. A mesma regra se aplica, basta utilizar entre chaves:

```
const Titulo = ({ texto, numero, pais }) => (
  <h1>{texto} <span className={pais}>{numero}</span> - <strong>{pais}</strong>
</h1>
);

function App() {
  return (
    <div id="container">
      <Titulo texto="Meu titulo" numero={30} pais="Brasil" />
      <p>Meu texto <span>destacado</span></p>
       <button type="button" id="btn">Aqui</button>
    </div>
  );
}
```

Fonte: autoral.

Podemos concatenar um valor dinâmico com string dentro de um atributo. Observe o *className* do elemento *span* do componente *Titulo*. Foi feita uma concatenação de string com variável/constante dentro do atributo.

```
const Titulo = ({ texto, numero, pais }) => (
  <h1>{texto} <span className={`meu-${pais}`}>{numero}</span> - <strong>{pais}
</strong></h1>
);

function App() {
  return (
    <div id="container">
      <Titulo texto="Meu titulo" numero={30} pais="Brasil" />
      <p>Meu texto <span>destacado</span></p>
       <button type="button" id="btn">Aqui</button>
    </div>
  );
}
```

Fonte: autoral.

Essa concatenação é feita igualmente quando feita com JavaScript puro utilizando template string. A única diferença é que é envolvida por chaves.

Condicionais

Podemos utilizar os operadores lógicos e operador ternário dentro do JSX.

A regra é a mesma de renderização de valores dinâmicos. Sempre que for por uma expressão JavaScript dentro do JSX, você precisa por entre chaves.

Caso não seja dentro do JSX, você até pode usar o if else normalmente.

Operador ternário

Renderizando o valor que depende de uma expressão. Veja o operador ternário entre chaves

```
const Pessoa = ({ texto, numero }) => (
  <h1>{texto} - {numero > 18 ? 'Maior' : 'Menor'}</h1>
);
```

Fonte: autoral.

Podemos retornar tags e outros componentes na expressão ternária

```
function Maior() {
  return <strong>Maior</strong>
}

const Pessoa = ({ texto, numero }) => (
  <h1>{texto} - {numero > 18 ? <Maior /> : 'Menor'}</h1>
);
```

Fonte: autoral.

If else

Caso não seja dentro do JSX, você pode usar um *if* normalmente

```
const Pessoa = ({ texto, numero }) => {
  if (numero === 18) {
    return <h3>Número sem componente</h3>
  }

  return (
    <h1>{texto} - {numero > 18 ? <Maior /> : 'Menor'}</h1>
  )
};
```

Fonte: autoral.

Listas

Uma coisa mais do que comum em uma aplicação web é querermos renderizar o conteúdo contido em um array, como listas. Podem ser listas normais, tabela, sequência de cards...

Vamos dizer que temos uma coleção (array de objetos) e queremos renderizar ela dentro de uma lista não ordenada sem precisar fazer isso na mão, item a item. Queremos fazer dinamicamente.

```
const todoItems = [
  { id: '111', title: 'Configurar projeto' },
  { id: '222', title: 'Embedar React' },
  { id: '333', title: 'Criar componentes' },
  { id: '444', title: 'Escrever testes' }
];
```

Fonte: autoral.

Para transformar esse array de objetos em uma lista de componentes, basta utilizar o map, que é uma função JavaScript para array, transformando uma lista de objetos em uma lista de componentes.

Como vamos utilizar o map dentro do JSX, vamos precisar escrever entre chaves:

```
function Items() {
  const todoItems = [
    { id: '111', title: 'Configurar projeto' },
    { id: '222', title: 'Embedar React' },
    { id: '333', title: 'Criar componentes' },
    { id: '444', title: 'Escrever testes' }
  ];

  return (
    <ul>
      {todoItems.map(item => (
         <li>{item.title}</li>
      ))}
    </ul>
  )
}
```

Fonte: autoral.

Podemos criar um componente apartado para o item renderizado:

```jsx
function Item(props) {
  return <li>{props.title}</li>
}

function Items() {
  const todoItems = [
    { id: '111', title: 'Configurar projeto' },
    { id: '222', title: 'Embedar React' },
    { id: '333', title: 'Criar componentes' },
    { id: '444', title: 'Escrever testes' }
  ];

  return (
    <ul>
      {todoItems.map(todoItem => (
        <Item id={todoItem.id} title={todoItem.title} />
      ))}
    </ul>
  )
}
```

Fonte: autoral.

Agora veja no console que ele reclama que falta uma prop key única em cada item da lista.

Isso acontece devido ao algoritmo de virtual dom do React, que só atualiza na tela os elementos que sofreram modificações.

Quando temos uma lista de itens semelhantes, fica confuso para o algoritmo do React identificar quais elementos sofreram modificações, quais foram removidos ou adicionados. Para ajudar, precisamos passar um identificador único para cada item da lista, esse identificador é passado pelo atributo *key*.

Podemos fazer da seguinte forma reutilizando o *id* de cada item do todo. Você também pode usar o próprio index da iteração se preferir.

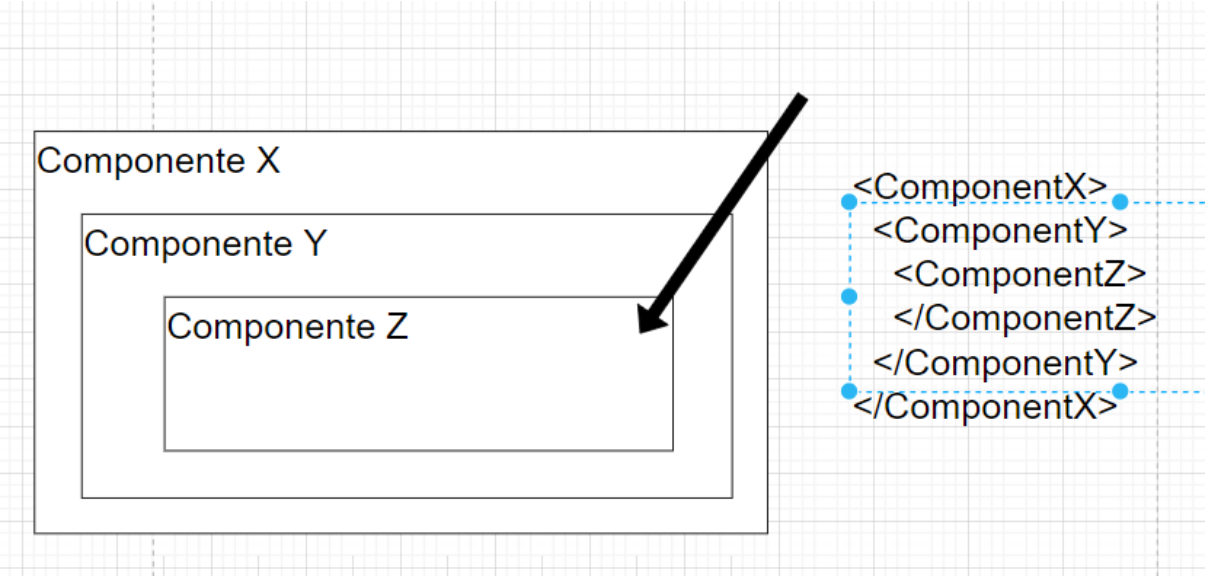
```
function Items() {
  const todoItems = [...];

  return (
    <ul>
      {todoItems.map(todoItem => (
        <Item key={todoItem.id} id={todoItem.id} title={todoItem.title} />
      ))}
    </ul>
  )
}
```

Fonte: autoral.

Como aplicar na prática o que aprendeu

Desenvolver com React significa desenvolver baseado em componentes. Quase tudo que se cria é um componente. Então, as coisas podem ficar complexas dependendo do cenário.

Imagina um monte de componentes utilizados para criar uma página/aplicação. Como você passa valores entre os componentes? O valor pode ser gerado em um componente, mas você precisa passar ele para um componente aninhado, como fazer?



Fonte: autoral.

Na imagem acima, precisamos passar um valor do componente mais alto nível para o mais aninhado.

O mais comum é fazer *prop drilling*, que basicamente significa passar de componente para componente até chegar no desejado. No exemplo abaixo, precisamos passar a propriedade *name* para o *ComponenteZ* a partir do componente *App:*

```
function ComponenteX(props) {
  return (
    <div className="componenteX">
      <ComponenteY name={props.name} />
    </div>
  )
}

function ComponenteY(props) {
  return (
    <div className="componenteY">
      <ComponenteZ name={props.name} />
    </div>
  )
}

function ComponenteZ(props) {
  return (
    <div className="componenteZ">
      <span>{props.name}></span>
    </div>
  )
}

function App() {
  const name = 'Ayrton';
  return <ComponenteX name={name} />
}
```

Fonte: autoral.

Observe que propagamos de componente por componente até chegar ao destino, que é o *ComponenteZ. Isso* é prop drilling. No *App* passamos para *ComponenteX*, dele passamos para *ComponenteY* e, finalmente, do *ComponenteY* para *ComponenteZ*.

Conteúdo bônus

descomplica

Tópicos avançados

Children Component

O React permite componentes do tipo filho. Até agora criamos componentes de auto-fechamento, ou seja, eles não recebem componente filho.

Você pode utilizar *Children Component* para algumas estratégias, uma estratégia interessante é para facilitar a passagem de valores entre componentes aninhados. Caso você passe um elemento filho para o componente, automaticamente ele recebe esse elemento através da propriedade *children*.

```
function ComponenteX(props) {
  return (
    <div className="componenteX">
      {props.children}
    </div>
  )
}

function ComponenteY(props) {
  return (
    <div className="componenteY">
      {props.children}
    </div>
  )
}

function ComponenteZ(props) {
  return (
    <div className="componenteZ">
      <span>{props.name}></span>
    </div>
  )
}

function App() {
  const name = 'Ayrton';
  return (
    <ComponenteX>
      <ComponenteY>
        <ComponenteZ name={name} />
      </ComponenteY>
    </ComponenteX>
  )
}
```

Fonte: autoral.

Observe que no código acima em nosso componente *App*, conseguimos passar o valor *name* diretamente de *App* para *ComponenteZ*, graças ao *Children Component* implementado nos componentes *ComponenteX* e *ComponenteY*.

Manipulação de eventos

Uma das funcionalidades básicas do desenvolvimento web com JavaScript, é a manipulação de eventos dos elementos HTML.

Você pode adicionar um evento de clique em um botão, evento de resize do navegador, evento de scroll e muito mais.

Para adicionar ouvintes de eventos nos seus elementos React, é muito semelhante ao HTML com JavaScript puro, você adiciona diretamente em seu elemento React como propriedade. Os nomes das propriedades de evento sempre são o prefixo ‘on’ + nome do evento (camelcase), exemplos: onClick, onKeyPress, onResize, onScroll, etc.

```
export const Eventos = () => {
  const ouvinteFuncao = () => {
    console.log('clicou')
  };
  return (
    <button type="button" onClick={ouvinteFuncao}>Clique</button>
  )
}
```

Fonte: autoral.

No código acima podemos ver o elemento *button* que recebe a propriedade *onClick*. Toda propriedade de evento recebe uma função, que será executada sempre que o evento acontecer. No caso acima, sempre que o botão for clicado.

A função de evento também recebe o objeto evento:

```
export const Eventos = () => {
  const ouvinteFuncao = (event) => {
    console.log(event)
  };
  return (
    <button type="button" onClick={ouvinteFuncao}>Clique</button>
  )
}
```

Fonte: autoral.

Veja que o resultado é um *SyntheticEvent*. Nada mais é que uma abstração sobre o objeto original do navegador de quando você faz pelo *addEventListener*. Ele é idêntico ao evento original, só que com algumas “normalizações” e compatibilidade entre navegadores, então você pode utilizar ele normalmente. Veja que se eu trocar o button por uma tag <a> e fizer o apontamento para um link, o comportamento padrão vai ser acessar o link.

```
export const Eventos = () => {
  const ouvinteFuncao = (event) => {
    console.log(event)
  };
  return (
    <a
      href="https://descomplica.com.br"
      onClick={ouvinteFuncao}>Descomplica</a>
  )
}
```

Fonte: autoral.

Para prevenir o comportamento padrão de acessar o link, fazemos da mesma forma que é com JavaScript puro, utilizando o método *preventDefault* do objeto *event:*

```
const ouvinteFuncao = (event) => {
  event.preventDefault();
  console.log(event)
};
```

Fonte: autoral.

Eventos em componentes de classes

Se nosso componente for uma classe, muitas vezes temos que fazer um trabalho a mais. Vamos transformar nosso componente Eventos em uma classe:

```
import React from "react";

export class Eventos extends React.Component {
  ouvinteFuncao(e) {
    e.preventDefault();
    console.log('clicou', e)
  };

  render() {
    return (
      <a
        href="https://descomplica.com.br"
        onClick={this.ouvinteFuncao}>Descomplica</a>
    )
  }
}
```

Fonte: autoral.

Veja que funcionou normalmente.

No JSX colocamos *render* e *ouvinteFuncao* como métodos da classe.

Essa abordagem funcionou perfeitamente, porém, existe o seguinte ponto de atenção: se tivéssemos que acessar um outro método da classe dentro do método que manipula o evento, não teria como, pois o método *ouvinteFuncao* perde a referência de *this* ao ser invocado como ouvinte de evento.

Faça o seguinte teste, adicione o console.log(this) dentro do método *ouvinteFuncao*, veja que o valor de *this* não é a referência para a própria classe.

```
ouvinteFuncao(e) {
  e.preventDefault();
  console.log('clicou', this)
}
```

Fonte: autoral.

Esse comportamento geralmente não é o desejado. O que queremos é que *this* “olhe” para a própria classe para caso precisemos acessar outras propriedades e métodos dela.

A forma de resolver isso é forçando o método a ter seu *this* apontando para a classe novamente. Podemos fazer em algumas partes do nosso código, vamos corrigir isso na própria propriedade de ouvinte de evento:

```
render() {
  return (
    <a
      href="https://descomplica.com.br"
      onClick={this.ouvinteEvento.bind(this)}>Descomplica</a>
  )
}
```

Fonte: autoral.

Utilizando *bind* conseguimos forçar o método *ouvinteEvento* a ter seu *this* como referência para a própria classe. Obs: essa forçação de *this* não é específica do React e sim do próprio JavaScript.

Referência Bibliográfica

SILVA, M. S. **React** - Aprenda Praticando: Desenvolva Aplicações web Reais com uso da Biblioteca React e de Seus Módulos Auxiliares. Novatec Editora, 2021.

STEFANOV, S. **Primeiros passos com React:** Construindo aplicações web, Novatec Editora, 2019.

Exercícios

1. Como passamos valores dinâmicos através de propriedades ao componente com sintaxe JSX que queremos renderizar? Vamos supor que queremos passar a variável “nomePessoa”

a) Utilizando entre chaves, ex: < MeuComponente valor={nomePessoa} />
b) Utilizando entre colchetes, ex: < MeuComponent valor=[nomePessoa] />
c) Como string normal, ex: < MeuComponent valor=”nomePessoa” />
d) Utilizando entre colchetes seguidos de string
e) Utilizando entre string e finalizados em colchetes

2. Em projetos grandes em React, temos o desafio da passagem de valores entre componentes muito aninhados, de cima para baixo (one-way data flow). Para isso, React oferece algumas soluções, uma dessas abordada até então, é:

a) Possibilidade de passar função como propriedade
b) Children component
c) Componente de alto fechamento
d) Listas dinâmicas com map
e) Utilizando condicionais

3. Para mesclarmos JSX com JavaScript, podemos apenas fazer uso de funcionalidades da linguagem que são expressões com retorno de valor. Qual dos seguintes recursos da linguagem não é possível utilizar dentro do JSX?

a) map e filter (métodos de array)

b) Operador ternário

c) Operadores lógicos no geral (&&, ||, !)

d) if else

e) map e &&

4. Qual a forma correta de definir um evento para um componente/elemento React?

a) Utilizando o método de elemento DOM addEventListener: MeuElementoDOM.addEventListener('click',…)

b) Utilizando o nome do evento com prefixo "on" na propriedade do elemento: < MeuComponente onClick={handleClick} />

c) Utilizando o nome do evento como propriedade do elemento: < MeuComponente click={handeClick} />

d) Passando o nome do evento com prefixo "on" direto para função do componente: MeuComponente.onClick = handleClick

e) Passando o nome do evento direto para função do componente MeuComponente.onClick

Gabarito

1. Letra A. Todo valor dinâmico que precisa ser renderizado dentro do JSX e passado via propriedade fica entre chaves. É a forma que o React utilizou para identificar valores dinâmicos sem conflitar com a sintaxe do React.

2. Letra B. Com seu componente aceitando um componente como filho, você tem acesso ao componente que esse filho também tem, com isso você consegue passar uma informação direto do primeiro componente para seu neto.

3. Letra D. If else não é uma expressão JavaScript, é um statement. Uma expressão é uma sintaxe que é avaliada pelo JavaScript e tem um valor retornado imediatamente, coisa que não acontece com if else.

4. Letra B. A forma de passar qualquer evento para um componente/elemento é através de propriedade, passando o nome do evento com prefixo “on”.