Objetivos:

I. Vantagens da composição de componentes;
II. Props;
III. Children como props.

I. Vantagens da composição de componentes

No React, a construção de interfaces com componentes reutilizáveis é uma prática que traz diversas vantagens para o desenvolvimento de aplicações modernas:

1) Reutilização de código: componentes permitem reutilizar trechos de código em diferentes partes da aplicação, reduzindo duplicação e facilitando a manutenção;
    - Ex.: um botão ou campo de entrada pode ser criado uma vez e reutilizado em vários contextos.
2) Manutenção simplificada: alterações em um componente afetam todas as suas instâncias automaticamente.
    - Se precisarmos atualizar o comportamento de um componente, precisamos fazer a mudança em um único lugar.
3) Composição de componentes: componentes podem ser combinados como blocos, como peças de LEGO, para formar componentes maiores e mais complexos.
    - Um componente pai pode conter componentes filhos, criando hierarquias claras.
4) Isolamento (encapsulamento): cada componente tem seu próprio estado e lógica encapsulados.
    - As alterações em um componente não afetam diretamente outros componentes, desde que não compartilhem estados.
5) Facilidade de testes: componentes menores e independentes são mais fáceis de testar individualmente (testes unitários);
6) Escalabilidade: uma interface construída com componentes facilita o escalonamento da aplicação.
    - À medida que a aplicação cresce, novos componentes podem ser adicionados sem refatorações significativas.
7) Melhora a legibilidade e organização: o uso de componentes torna o código mais legível e organizado, facilitando a compreensão por outros desenvolvedores.
    - Cada componente tem uma única responsabilidade, seguindo o princípio do SRP (Single Responsibility Principle).
8) Renderização eficiente com o Virtual DOM: o React utiliza um mecanismo chamado Virtual DOM, que otimiza a renderização dos componentes.
    - Apenas os componentes que realmente sofreram alterações são atualizados no DOM real.
9) Maior produtividade da equipe: equipes podem trabalhar paralelamente em componentes diferentes, acelerando o desenvolvimento.

II. Props

No React, props (abreviação de "properties") são objetos usados para passar dados de um componente pai para um componente filho. No exemplo a seguir App é o componente pai e Message é o componente filho.

```
export default function App() {
  return (
    <>
      <Message firstname="Ana" lastname="Gouvea" />
      <Message firstname="Mônica" lastname="Silva" />
      <Message firstname="Rodrigo" lastname="Souza" />
    </>
  )
}

function Message(props:Props){
  return <h3>Olá {props.firstname} {props.lastname}!</h3>;
}

interface Props {
  firstname: string;
  lastname: string;
}
```

As propriedades recebidas pelo componente Message são especificadas pelo tipo de dado definido na interface Props. Desta forma, a chamada do componente <Message> precisa ter exatamente as propriedades firstname e lastname.

As chamadas a seguir estão incorretas:

- <Message /> - falta as propriedades firstname e lastname;
- <Message firstname="João" /> - falta a propriedade lastname;
- <Message lastname="Carvalho" /> - falta a propriedade firstname;
- <Message firstname="João" lastname="Carvalho" age="20" /> - a propriedade age não faz parte do tipo Props definido na interface;
- <Message firstname={10} lastname={20} /> - o tipo number não é atribuível ao tipo string definido na interface Props.

As propriedades firstname e lastname podem ser desestruturadas ao serem recebidas no componente filho:

```
function Message({firstname,lastname}:Props){
  return <h3>Olá {firstname} {lastname}!</h3>;
}
```

No código a seguir, a instrução <Message {...person} /> desestrutura e repassa automaticamente as propriedades do objeto person como props para o componente Message. Isso torna o código mais conciso e fácil de manter.

```
export default function App() {
  const person = {
    firstname: "José",
    lastname: "Afonso"
  };

  return (
    <>
      <Message firstname="Ana" lastname="Gouvea" />
      <Message firstname="Mônica" lastname="Silva" />
      <Message firstname="Rodrigo" lastname="Souza" />
      <Message {...person} />
    </>
  )
}
```

O fluxo é unidirecional (One-Way Data Binding), ou seja, os dados fluem sempre do componente pai para o componente filho por meio das props. O componente filho nunca altera diretamente os dados do componente pai, mas pode solicitar alterações por meio de funções passadas como props. No código a seguir, o componente Contador recebe como props:

- A propriedade value, que representa o valor atual do contador;
- A função setValue, responsável por atualizar o estado do contador;
- A função reset, que redefine o contador para zero.

Essas funções permitem que o componente filho (Contador) solicite alterações no estado do componente pai (App) de forma controlada.

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

export default function App() {
  const [count, setCount] = useState(0);

  function reset() {
    setCount(0);
  }

  return (
    <>
      <h4>Contador: {count}</h4>
      <Contador value={count} setValue={setCount} reset={reset} />
    </>
  );
}

function Contador(props: CountProps) {
  return (
    <div>
```

```
      <button onClick={() => props.setValue((prev) => prev + 1)}>
        Incrementar
      </button>
      <button onClick={props.reset}>Resetar</button>
    </div>
  );
}

interface CountProps {
  value: number;
  setValue: (value: number | ((prev: number) => number)) => void;
  reset: () => void;
}
```

Observações:

- As propriedades (props) são imutáveis dentro do componente filho. Isso significa que o componente filho não pode alterar diretamente o valor de uma propriedade recebida;
- As props representam dados de leitura que são controlados exclusivamente pelo componente pai. O componente filho pode apenas solicitar alterações por meio de funções de callback recebidas como props, como `setValue` e `reset` no exemplo;
- Essa restrição garante que o fluxo de dados permaneça unidirecional, prevenindo comportamentos inesperados e mantendo a previsibilidade e consistência do estado da aplicação;
- No código anterior, o componente filho (`Contador`) não altera diretamente a propriedade `value`, mas utiliza as funções `setValue` e `reset` fornecidas pelo componente pai (App) para modificar o estado associado;
- Esse padrão promove uma clara separação de responsabilidades entre os componentes, mantendo o controle do estado no componente pai e delegando ao filho apenas a responsabilidade de disparar ações específicas.

III. Children como props

No React, `children` é uma propriedade especial usada para representar o conteúdo que é passado entre as tags de abertura e fechamento de um componente. Essa propriedade permite que um componente seja mais flexível, possibilitando a inclusão de elementos JSX, texto, ou até mesmo outros componentes como filhos.

No código a seguir, passamos diferentes elementos HTML entre as tags `<Message>` e `</Message>`.

```
export default function App() {
  return (
    <>
      <Message>
        <h3>Boa noite!</h3>
        <p>Sem bem-vindos ao curso.</p>
      </Message>
```

```
      <Message>
        <div>Dias de aula:</div>
        <ol>
          <li>Terça-feira</li>
          <li>Quarta-feira</li>
        </ol>
      </Message>
    </>
  );
}

function Message({ children }: Props) {
  return <div style={messageStyle}>{children}</div>;
}

interface Props {
  children: React.ReactNode;
}

const messageStyle = {
  display: "inline-block", // garante que o elemento se ajuste ao conteúdo
  width: "auto", // largura se adapta automaticamente
  margin: "10px",
  border: "1px solid black",
  padding: "10px",
  whiteSpace: "nowrap", // impede quebras de linha indesejadas
};
```

`React.ReactNode` é um tipo amplo que inclui qualquer coisa que o React pode renderizar: strings, números, elementos JSX, fragmentos, ou até mesmo `null` e `undefined`.

IV. Exercícios

Veja os vídeos se tiver dúvidas nos exercícios:

Exercícios 1 a 3 - https://youtu.be/Vn3e6K63K2g

Exercícios 4 a 6 - https://youtu.be/aqxL-lQbzXM

Exercício 1 – Adicione o componente Button no código a seguir para responder ao clique.


```
import { CSSProperties, useState } from "react";

export default function App() {
  const [count, setCount] = useState(0);

  function increment() {
```

```
    setCount((prev) => prev + 1);
  }

  function decrement() {
    setCount((prev) => prev - 1);
  }

  return (
    <>
      <h4>Valor: {count}</h4>
      <div>
        <Button action={increment} label="Incrementar" />
        <Button action={decrement} label="Decrementar" />
      </div>
    </>
  );
}

const buttonStyle: CSSProperties = {
  backgroundColor: "#567196",
  color: "white",
  padding: "10px 20px",
  marginRight: "10px",
  fontSize: "16px",
  fontWeight: "bold",
  border: "none",
  borderRadius: "5px",
  cursor: "pointer",
};
```

Exercício 2 – Altere o componente Button para receber o corpo do botão como children. Utilize os códigos a seguir para estilizar os botões.

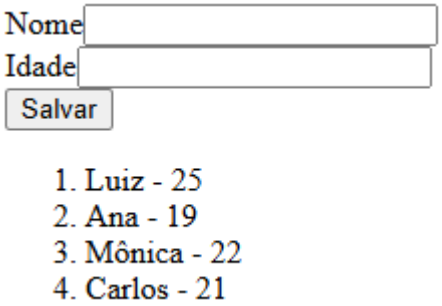

```
  return (
    <>
      <h4>Valor: {count}</h4>
      <div style={{display:"flex"}}>
        <Button action={increment}>
          <img
            src="https://cdn-icons-png.flaticon.com/512/1828/1828926.png"
            alt="Incrementar"
            style={iconStyle}
          />
          <span>Incrementar</span>
        </Button>
        <Button action={decrement}>
```

```
          <img
            src="https://cdn-icons-png.flaticon.com/512/1828/1828906.png"
            alt="Decrementar"
            style={iconStyle}
          />
          <span>Decrementar</span>
        </Button>
      </div>
    </>
  );

const buttonStyle: CSSProperties = {
  backgroundColor: "#567196",
  color: "white",
  padding: "10px 20px",
  marginRight: "10px",
  fontSize: "16px",
  fontWeight: "bold",
  border: "none",
  borderRadius: "5px",
  cursor: "pointer",
  display: "flex", // flexbox para alinhamento
  alignItems: "center", // alinha ícone e texto verticalmente
  gap: "8px", // espaçamento entre ícone e texto
};

const iconStyle: CSSProperties = {
  width: "20px",
  height: "20px",
};
```

Exercício 3 – Complete o código a seguir codificando os componentes Form e List.




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

export default function App() {
  const [users, setUsers] = useState<User[]>([]);

  function add(user:User) {
```

```
    setUsers((prev) => [...prev, user]);
  }

  return (
    <>
      <Form add={add} />
      <List users={users} />
    </>
  );
}

interface User {
  name: string;
  age: string;
}
```

Exercício 4 – Utilizando a função remove a seguir, inclua no código do Exercício 3 a opção para excluir o elemento ao clicar com o botão direito do mouse.

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

export default function App() {
  const [users, setUsers] = useState<User[]>([]);

  function add(user: User) {
    setUsers((prev) => [...prev, user]);
  }

  function remove(index: number) {
    setUsers(users.filter((_, i) => i !== index));
  }

  return (
    <>
      <Form add={add} />
      <List users={users} remove={remove} />
    </>
  );
}

function Form({ add }: FormProps) {
  const [name, setName] = useState("");
  const [age, setAge] = useState("");

  return (
    <div>
      <div>
```

```
        <label htmlFor="name">Nome</label>
        <input
          id="name"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </div>
      <div>
        <label htmlFor="age">Idade</label>
        <input id="age" value={age} onChange={(e) => setAge(e.target.value)} />
      </div>
      <div>
        <button onClick={() => add({ name, age })}>Salvar</button>
      </div>
    </div>
  );
}
```

Exercício 5 – Codifique o aplicativo ao lado utilizando o componente Button a seguir. Ao clicar no botão o texto da fonte deverá ser alterado para refletir a origem do clique.


```
function Button({label, set}: Props) {
  return <button style={buttonStyle} onClick={() =>set(label)}>{label}</button>;
}

interface Props {
  label: string;
  set: (value:string) => void;
}

const buttonStyle: CSSProperties = {
  backgroundColor: "#567196",
  color: "white",
  padding: "10px 20px",
  marginRight: "10px",
  fontSize: "16px",
  fontWeight: "bold",
  border: "none",
  borderRadius: "5px",
  cursor: "pointer",
};
```

Exercício 6 – Altere o código do Exercício 5 para mostrar os botões clicados em uma lista. Essa lista deverá estar em um componente e ser chamado no componente App.

Fontes:

1. B
2. D
3. A
4. C
5. B