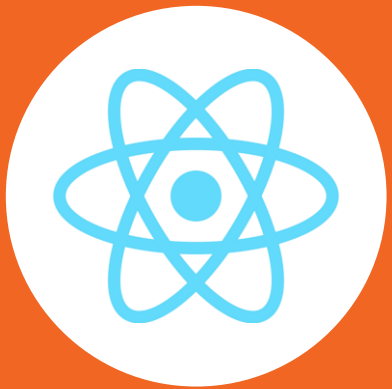
FPT Education

FPT POLYTECHNIC



LẬP TRÌNH ĐA NỀN TẢNG VỚI REACT NATIVE

BÀI 2: GIỚI THIỆU VỀ HOOKS

PHẦN 1: GIỚI THIỆU VỀ HOOKS, useState, useRef, userEffect

https://caodang.fpt.edu.vn/

FPT Education
FPT POLYTECHNIC

Mục tiêu

- Giới thiệu chung về **hooks** trong React Native
- Biết cách sử dụng **useState**
- Biết cách sử dụng **useRef**
- Biết cách sử dụng **useEffect**

FPT Education
FPT POLYTECHNIC

Giới thiệu hooks

- Giới thiệu chung về **Hooks** trong React Native:

  Các Hooks cho phép sử dụng các tính năng khác nhau của React. Bạn có thể sử dụng các Hooks có sẵn hoặc kết hợp chúng để xây dựng các Hooks của riêng mình.

- Có 5 loại **Hooks** chính thường được sử dụng:

  - **1. State Hooks**

    Dùng để lưu trữ giá trị cho component, giá trị state có thể thay đổi, mỗi lần biến state này thay đổi thì sẽ làm **re-render** component chứa nó.

Sử dụng useState hook để lưu trữ giá trị:

```
const [count, setCount] = useState(0);
const [count2, setCount2] = useState(0);
```

2.Context Hooks

Context cho phép một component nhận thông tin từ các thành phần cha ở xa mà không cần chuyển nó qua **props**. Ví dụ, thành phần cấp cao nhất của ứng dụng có thể truyền giá trị nó đến tất cả các thành phần bên dưới, bất kể độ sâu của chúng.

useContext đọc và đăng ký theo dõi một context.

3.Ref Hooks

Ref dùng để lưu trữ lại giá trị của biến, việc cập nhật lại giá trị của ref sẽ không gây **re-render** component.

Có thể giữ bất kỳ giá trị nào trong đó, nhưng thường được sử dụng để giữ một nút DOM.
Ngoài ra, sử dụng thêm **useImperativeHandle** cho phép thêm function, biến,... vào ref của component.

4.Effect Hooks

Effects dùng để lắng nghe việc thay đổi giá trị và re-render của component, hook này được sử dụng để quản lý life-cycle ứng dụng

```
useEffect(() => {
  prevCount.current = count;
}, [count]);
```

- **useLayoutEffect** kích hoạt trước khi ứng dụng khởi tạo lại màn hình.

5.Performance Hooks

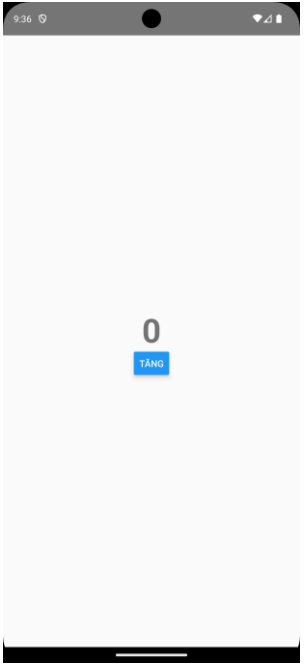
Giúp tối ưu hiệu suất ứng dụng, tránh việc re-render lại component không cần thiết.

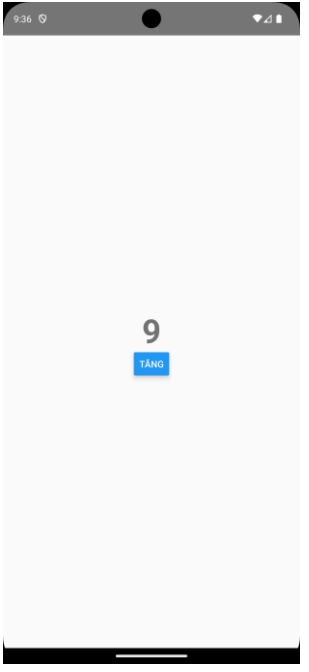
Những hook này cho React biết để sử dụng lại một phép tính được lưu vào bộ nhớ cache hoặc để bỏ qua việc tính toán lại function hoặc componnent không cần thiết.

Giới thiệu hooks

- **useMemo** cho phép lưu vào bộ nhớ cache kết quả của một phép tính tốn kém.
- **useCallback** cho phép lưu một hàm vào bộ nhớ cache để tránh việc tạo lại chúng mỗi khi component re-render.
- **memo** là một Higher-Order Component (HOC) được sử dụng để tối ưu hóa hiệu suất bằng cách ghi nhớ kết quả render của một component. Điều này có nghĩa là React chỉ render lại component đó nếu có sự thay đổi trong các props của nó.

- Ví dụ: Sử dụng hook **useState**





- Code mẫu ở bên dưới

```
export const UseStateScreen = () => {
  const [count, setCount] = useState(0);

  const handleIncrease = () => {
    setCount(count + 1);
  };

  return (
    <View style={styles.container}>
      <Text style={styles.textCount}>{count}</Text>
      <Button title="Tăng" onPress={handleIncrease} />
    </View>
  );
};
```

- Có thể lưu bất cứ kiểu dữ liệu nào trong **useState,** dưới đây là một ví dụ về lưu biến object vào state bằng cách như sau:

```
export const UseStateScreen = () => {
  const [inforUser, setInforUser] = useState({
    name: 'Nguyen Van A',
    age: 20,
  });

  const updateInforUser = () => {
    setInforUser({
      ...inforUser,
      age: 21,
    });
  };
```

useEffect

- **useEffect** có 3 loại, không có dependencies, có dependencies nhưng rỗng, có dependencies

```
useEffect(() => {
  // console.log('useEffect này chạy mỗi lần component render');
});

useEffect(() => {
  // console.log('useEffect chỉ chạy lần đầu tiên khi component render');
}, []);

useEffect(() => {
   // console.log('useEffect khởi chạy khi count thay đổi giá trị');
}, [count]);
```

- Chạy lại ứng dụng, kết quả của log

```
LOG  Running "DaNenTang2" with {"rootTag":11}
LOG  useEffect này chạy mỗi lần component render
LOG  useEffect chỉ chạy lần đầu tiên khi component render
LOG  useEffect chạy mỗi lần state count thay đổi giá trị, count =  0
```

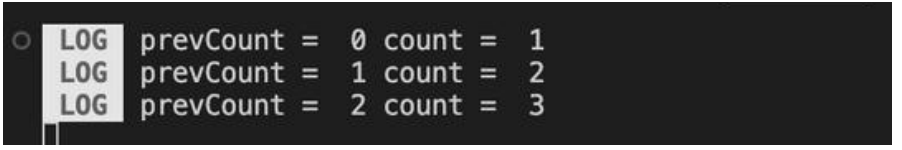
- Log sau khi tăng state **count** lên một giá trị

```
LOG  Running "DaNenTang2" with {"rootTag":11}
LOG  useEffect này chạy mỗi lần component render
LOG  useEffect chỉ chạy lần đầu tiên khi component render
LOG  useEffect chạy mỗi lần state count thay đổi giá trị, count =  0
LOG  useEffect này chạy mỗi lần component render
LOG  useEffect chạy mỗi lần state count thay đổi giá trị, count =  1
```

- Demo một số tính chất của **useRef**

```
const prevCount = useRef();

useEffect(() => {
  prevCount.current = count;
}, [count]);

console.log(
  'prevCount = ', prevCount.current, 'count = ', count
);
```

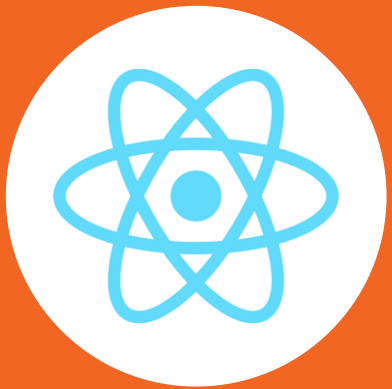
- Sau khi nhấn nút Tăng lần thứ 3, và đây là log của phần code phía trên



  - Giá trị của **preCount** và **count** không bằng nhau, bởi vì sau khi giá trị count được cập nhật, thì nó sẽ gây render lại component, sau đó log sẽ xuất ra và sau khi DOM đã được render xong thì **useEffect** mới được chạy vào.

FPT Education

FPT POLYTECHNIC



LẬP TRÌNH ĐA NỀN TẢNG VỚI REACT NATIVE

BÀI 2: GIỚI THIỆU VỀ HOOKS

PHẦN 2: GIỚI THIỆU VỀ useMemo, useMemo, memo, useContext

https://caodang.fpt.edu.vn/

Mục tiêu

- Tối ưu hoá hiệu năng ứng dụng bằng **useMemo**, **memo**, **useCallback**
- Chia sẽ dữ liệu bằng **useContext**

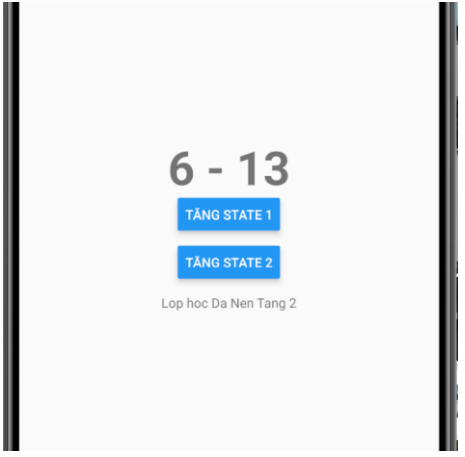
memo

- Trong React Native, **memo** là một React Hook dùng để tối ưu hiệu suất của component bằng cách tránh việc render lại không cần thiết khi không có sự thay đổi về props hoặc state của component.

  Khi một component được wrap bởi **memo**, React Native sẽ lưu lại phiên bản hiện tại của component và **so sánh** với phiên bản trước đó khi component được render lại. Nếu **props** hoặc **state** không thay đổi, component sẽ không được render lại mà sử dụng phiên bản đã lưu trữ để hiển thị.

Ví dụ: Sử dụng **memo**

Hàm xử lý




```
const UseMemoCountScreen() {
  const [count, setCount] = useState(0);
  const [count2, setCount2] = useState(0);

  const handleIncrease = () => {
    setCount(prev => prev + 1);
  };
  const handleIncrease2 = () => {
    setCount2(prev => prev + 1);
  };
```

Giao diện

```
  return (
    <View style={styles.container}>
      <Text style={styles.textCount}>
        {count} - {count2}
      </Text>
      <Button title="Tăng state 1" onPress={handleIncrease} />
      <View style={styles.seperate} />
      <Button title="Tăng state 2" onPress={handleIncrease2} />
      <Content count={count} />
    </View>
  );
}
```

- Tạo component con **Content** được bọc bằng **memo**

```
export const Content = memo(({count} = {}) => {
  console.log('re-render in Content, count =  ', count);
  return (
    <View style={styles.container}>
      <Text>Lop hoc Da Nen Tang 2</Text>
    </View>
  );
});
```

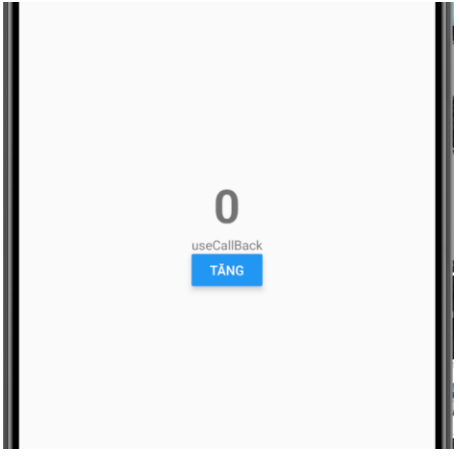
- Truyền state **count** vào **Content** component, khi thay đổi giá trị của state **count2** thì **Content** component sẽ không bị re-render.

useCallback

- Trong React Native, **useCallback** là một hook được sử dụng để tối ưu hóa performance của các component bằng cách tránh việc rendering lại một hàm hoặc **callback** function mỗi khi component được render lại.

  Khi một component được render lại, các hàm và **callback** functions trong component cũng sẽ được tạo lại, dẫn đến việc tốn tài nguyên và giảm performance của ứng dụng. **useCallback** giúp giải quyết vấn đề này bằng cách "**nhớ lại**" hàm hoặc callback function và chỉ tạo lại chúng khi các dependencies của nó thay đổi.

- Ví dụ: Sử dụng **useCallback**




```
export const UseCallBackScreen = () => {
  const [count, setCount] = useState(0);

  const handleIncrease1 = useCallback(() => {
    setCount(prevCount => prevCount + 1);
  }, []);

  return (
    <View style={styles.container}>
      <Text
style={styles.textCount}>{count}</Text>
      <ContentUseCallBack
onIncrease={handleIncrease1} />
    </View>)};
```

useCallback

- Tạo component con **ContentUseCallBack** được bọc bằng **memo**

```
export const ContentUseCallBack = memo(({onIncrease}) => {
  console.log('re-render');
  return (
    <View>
      <Text>useCallBack</Text>
      <Button title="Tăng" onPress={onIncrease} />
    </View>
  );
});
```

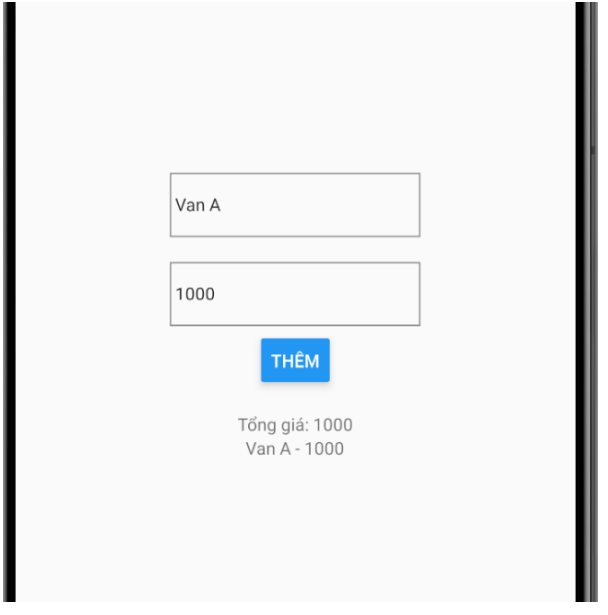
Component **ContentUseCallBack** không re-render lại, mặc dù có truyền hàm **onIncrease**, bởi vì hàm này được được bọc bởi **useCallBack.**

useMemo

- **useMemo** là một hook trong React để tối ưu hóa việc tính toán các giá trị phức tạp và tránh việc tính toán lặp đi lặp lại không cần thiết trong quá trình **render** của components.

  Khi sử dụng **useMemo**, bạn có thể chỉ định một hàm và một danh sách các **dependencies** (phụ thuộc) và RN sẽ gọi hàm này và trả về kết quả của nó. Kết quả này được lưu trữ trong bộ nhớ đến khi các **dependencies** thay đổi. Nếu **dependencies không thay đổi**, kết quả được lưu trữ sẽ được tái sử dụng cho các render tiếp theo, giúp giảm thiểu việc tính toán không cần thiết.

- Ví dụ: Sử dụng **useMemo**



useMemo

- Hàm xử lý

```
export default function UseMemoScreen() {
  const [name, setName] = useState('');
  const [price, setPrice] = useState('');
  const [products, setProducts] = useState([]);

  const handleSubmit = () => {
    setProducts([...products, {name, price: +price}]);
  };
  const total = useMemo(() => {
    console.log('Tính toán lại ...');
    const result = products?.reduce((_result, prod) => {
      return _result + prod.price;
    }, 0);
    return result;
  }, [products]);
```

Hàm trong **useMemo** chỉ render lại khi state **products** thay đổi giá trị

- Giao diện

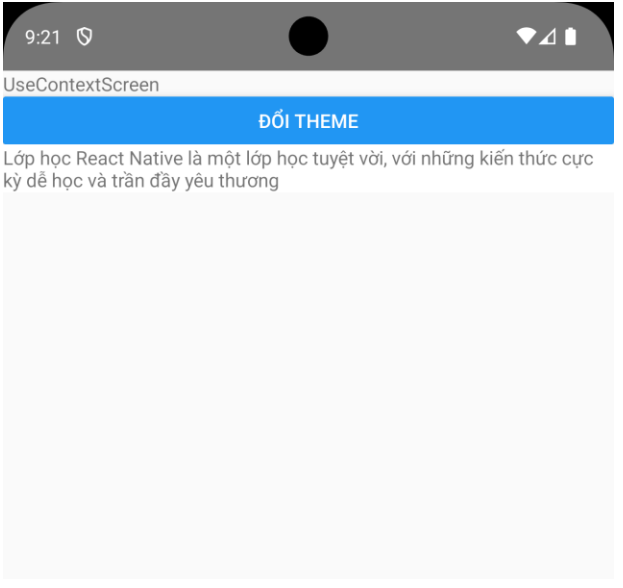
```
return (
  <View style={styles.container}>
    <TextInput
      value={name}
      ... />
    <TextInput .... />
    <Button title="Thêm" onPress={handleSubmit} />
    <Text style={styles.sumPrice}>Tổng giá: {total}</Text>
    {products?.map((product, index) => (
      <Text key={index}>
        {product.name} - {product.price}
      </Text>
    ))}
  </View>);
```

useContext

- **useContext** là một hook trong React giúp truy cập vào các giá trị được chia sẻ trên toàn bộ ứng dụng thông qua context, mà không cần truyền qua các components con.

  Context là một cơ chế cho phép chia sẻ dữ liệu giữa các component trong cây component mà không cần truyền props qua từng component. Các giá trị context có thể được cung cấp bởi một component ở mức cao nhất trong cây component và các component con có thể truy cập vào chúng thông qua **useContext**.

- Ví dụ: Sử dụng **useContext**



useContext

- Bọc **Provider** vào component cha vào nơi muốn sử dụng **useContext**

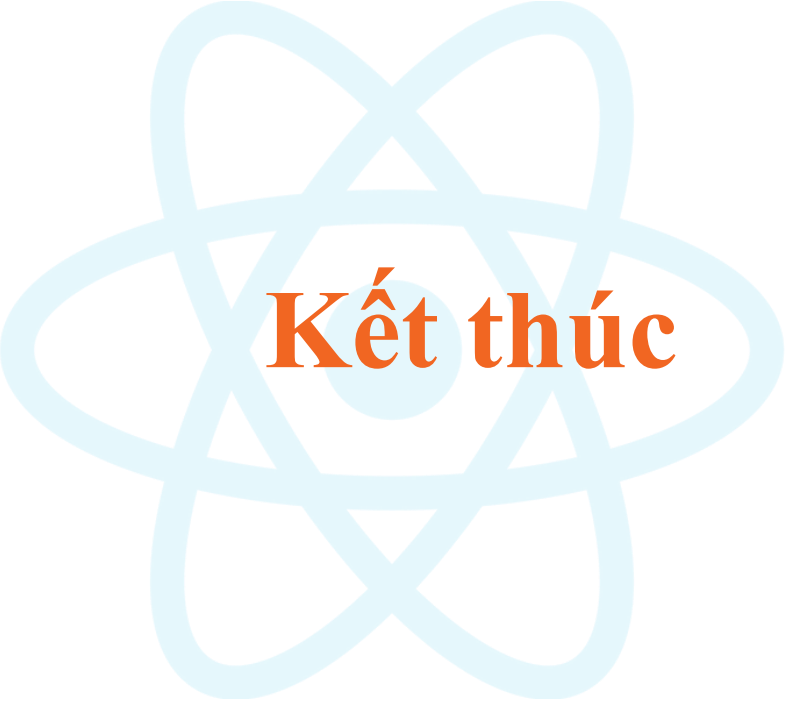
```
export const ThemeContext = createContext('light');
export default function UseContextScreen() {
  const [theme, setTheme] = useState('light');

  const toggleTheme = () => {
    setTheme(theme === 'dark' ? 'light' : 'dark');
  };
  return (
    <ThemeContext.Provider value={theme}>
      <View>
        <Text>UseContextScreen</Text>
        <Button title="Đổi theme" onPress={toggleTheme} />
        <Paragraph />
      </View>
    </ThemeContext.Provider>)}
```

- **Paragraph** là component con bên trong component **UseContextScreen** cho nên có thể lấy giá trị bên trong **useContext**

```
export default function Paragraph() {
  const theme = useContext(ThemeContext);
  return (
    <View style={{backgroundColor: theme === 'light' ? 'white' :
'gray'}}>
      <Text>
        Lớp học React Native là một lớp học tuyệt vời, với những kiến
thức cực
        kỳ dễ học và trần đầy yêu thương
      </Text>
    </View>
  );
}
```

Tổng kết

- Hiểu và sử dụng các **hook** trong React Native
- Kết hợp sử dụng các **hook** với nhau
- Tối ưu hoá ứng dụng khi sử dụng hook
- Cải thiện performance bằng các **hook**
- Chia sẻ dữ liệu giữa các component bằng useContext



Kết thúc