React Native Study Notes: Components, State Hooks, and Props

Introduction to React Native

React Native is a framework for building mobile applications using JavaScript and React. It allows developers to create cross-platform apps for iOS and Android with a single codebase. This set of notes focuses on three core concepts: **Components**, **State Hooks**, and **Props**, which are essential for building dynamic and reusable React Native applications.

1. Components in React Native

Components are the building blocks of React Native applications. They represent reusable pieces of the user interface (UI), such as buttons, text inputs, or entire screens.

Types of Components

- **Functional Components**: Modern way to write components using JavaScript functions. They are simpler and support hooks.
- **Class Components**: Older approach using ES6 classes, less common with the advent of hooks but still supported.

Creating a Functional Component

Functional components are defined as JavaScript functions that return JSX (a syntax extension for JavaScript that resembles HTML). Below is an example of a simple functional component:

```
import React from 'react';
import { View, Text } from 'react-native';

const Greeting = () => {
  return (
    <View>
      <Text>Hello, React Native!</Text>
    </View>
  );
};

```

```
export default Greeting;
```

Listing 1: Simple Functional Component

Key Points

- Components must return a single JSX element (e.g., wrap multiple elements in a `View`).
- Use `export default` to make the component reusable in other parts of the app.
- React Native components like `View`, `Text`, and `Button` are imported from `react-native`.

2. Props in React Native

Props (short for properties) are used to pass data from a parent component to a child component. They make components reusable and dynamic.

Using Props

Props are passed as attributes to a component and accessed as an object in the component's function. Here's an example:

```
import React from 'react';
import { View, Text } from 'react-native';

const UserCard = (props) => {
  return (
    <View>
      <Text>Name: {props.name}</Text>
      <Text>Age: {props.age}</Text>
    </View>
  );
};

const App = () => {
  return (
    <View>
      <UserCard name="Alice" age={25} />
      <UserCard name="Bob" age={30} />
    </View>
  );
};

export default App;
```

Listing 2: Using Props in a Component

Key Points

- Props are read-only; a component cannot modify its own props.

- Props can be any JavaScript value: strings, numbers, objects, or even functions.
- Use default props to provide fallback values if props are not passed.

Default Props

You can define default props to ensure a component works even if props are not provided:

```
UserCard.defaultProps = {
  name: 'Guest',
  age: 18
};
```

Listing 3: Setting Default Props

3. State Hooks in React Native

State allows components to manage and update their own data, making them interactive. In functional components, state is managed using **hooks**, introduced in React 16.8.

The useState Hook

The `useState` hook is used to add state to functional components. It returns an array with two elements: the current state value and a function to update it.

```
import React, { useState } from 'react';
import { View, Text, Button } from 'react-native';

const Counter = () => {
  const [count, setCount] = useState(0);

  return (
    <View>
      <Text>Count: {count}</Text>
      <Button
        title="Increment"
        onPress={() => setCount(count + 1)}
      />
    </View>
  );
};

export default Counter;
```

Listing 4: Using useState Hook

Key Points

- `useState` takes an initial state value (e.g., `0` in the example).
- The setter function (e.g., `setCount`) updates the state and triggers a re-render.

- State updates are asynchronous; use functional updates for state dependent on previous state:

```
setCount(prevCount => prevCount + 1);
```

Listing 5: Functional Update with useState

Managing Complex State

For complex state (e.g., objects), ensure to spread the previous state to avoid overwriting:

```
import React, { useState } from 'react';
import { View, Text, Button } from 'react-native';

const Profile = () => {
  const [user, setUser] = useState({ name: 'Alice', age: 25 });

  const updateAge = () => {
    setUser({ ...user, age: user.age + 1 });
  };

  return (
    <View>
      <Text>Name: {user.name}, Age: {user.age}</Text>
      <Button title="Increase Age" onPress={updateAge} />
    </View>
  );
};

export default Profile;
```

Listing 6: Managing Object State

4. Combining Components, Props, and State

Here's a more complex example combining components, props, and state to create an interactive list:

```
import React, { useState } from 'react';
import { View, Text, Button, FlatList } from 'react-native';

const ListItem = ({ item }) => (
  <View>
    <Text>{item.text}</Text>
  </View>
);

const TodoList = () => {
  const [todos, setTodos] = useState([
    { id: '1', text: 'Learn React Native' },
    { id: '2', text: 'Build an app' }
```

```
  ]);

  const addTodo = () => {
    setTodos([
      ...todos,
      { id: Math.random().toString(), text: 'New Task' }
    ]);
  };

  return (
    <View>
      <Button title="Add Todo" onPress={addTodo} />
      <FlatList
        data={todos}
        renderItem={({ item }) => <ListItem item={item} />}
        keyExtractor={item => item.id}
      />
    </View>
  );
};

export default TodoList;
```

Listing 7: Interactive List with Components, Props, and State

5. Best Practices

- **Components**: Keep components small and focused on a single responsibility.
- **Props**: Use descriptive prop names and validate them using `PropTypes` (optional library).
- **State**: Avoid overusing state; use props for data that doesn't need to change within a component.
- **Reusability**: Design components to be reusable across different parts of the app.
- **Performance**: Use `FlatList` for long lists to optimize rendering.

6. Exercises for Practice

1. Create a component that displays a user's name and email, passed as props.
2. Build a counter app with buttons to increment, decrement, and reset the count using `useState`.
3. Create a todo list app where users can add and remove tasks using state and props.

Conclusion

Understanding components, props, and state hooks is fundamental to building React Native applications. Components define the UI, props enable data sharing, and state hooks make components interactive. Practice these concepts by building small projects to solidify your understanding.