# React Hooks

## Introduction

### What are hooks?

Hooks are a new addition in React 16.8. They are functions that let you manage the internal state of component and handle post rendering side effects.
**Note**: Before we continue, note you can try Hooks in a few components without rewriting any existing code and hooks are 100% backwards-compatible. They don't contain any breaking changes.

### Motivation to use hooks

Hooks can be used to address the following problems caused by using class based components:

1. **Hard to reuse stateful logic between components**
   Hooks provide a way to separate stateful logic from a component, which allows for independent testing and reusability. With Hooks, you can reuse this logic in multiple components without restructuring the component hierarchy. This flexibility makes it easy to share Hooks with others and promote community collaboration.

2. **Complex components become difficult to understand**
   In class-based components, related code often gets mixed with unrelated code, which can lead to bugs and inconsistencies. Hooks address this issue by allowing you to break down a component into smaller functions that handle related pieces of logic, like setting up a subscription or fetching data. This approach is more intuitive than forcing a split based on lifecycle methods.

3. **Classes can be confusing at times**

   Classes in React can be a significant hurdle for learning the framework. Not only do you have to understand how "this" works in JavaScript, which can be quite different from other languages, but they can also complicate code reuse and organization. Hooks provide a simpler way to use React's features without relying on classes. By embracing functions, Hooks allow for a more intuitive and functional programming style that is closer to the conceptual nature of React components.

## Rules of Hooks

The two main rules of Hooks in React are:

1. **Only call Hooks at the top level:**

   This rule states that Hooks should only be called at the top level of a function component or another custom Hook. They should not be called inside loops, conditions, or nested functions, as this can lead to unexpected behavior and bugs.

   **Example 1**: **Incorrectly** using hooks inside a function

```
const showAlert = () => {

      useEffect(() => {

        alert("Dependency has changed");

      }, [dependency]);

}
```

   **Example 2**: **Correctly** using hooks at the top level of the component

```
useEffect(() => {

     alert("Dependency has changed");

}, [dependency]);
```

2. **Only call Hooks from React function components:**

   This rule states that Hooks can only be used in React function components or other custom Hooks. They should not be used in class components or regular JavaScript functions, as this can cause errors or crashes.

   **Example 1**: **Incorrectly** using hooks inside a class based component

```
class Example extends Component{

      useEffect(() => {

        alert("Dependency has changed");

      }, [dependency]);

}
```

   **Example 2**: **Correctly** using hooks inside a functional component

```
const Example = () => {

      const [dependency, setDependency] = useState("");

      useEffect(() => {

        alert("Dependency has changed");

      }, [dependency]);

};
```

Adhering to these rules ensures that Hooks work as intended and can help to prevent common issues and errors when working with React.

## Order of Hooks

In React, hooks are executed in the order in which they are written. The order of Hooks is important. The order in which Hooks are called within a component must always be the same, so that React can correctly associate state and props with each Hook.

## State in Function & Class based components

State management in functional and class-based React components is essentially the same, but the syntax and implementation details differ.

### Syntax:

Class-based components have lifecycle methods, such as **componentDidMount** and **componentDidUpdate**, which allow for fine-grained control over the state and behavior of the component. Functional components have **hooks** that let you hook into the state and lifecycle features of the component but the syntax and implementation are different.

### Updates and Side effects:

In class-based components, state is updated via the **setState** method. In functional components with hooks, state is updated via the update function returned by the **useState** hook. Side effects in functional components are managed using the **useEffect** hook which is a replacement for the lifecycle methods used in class-based components, such as **componentDidMount** and **componentDidUpdate**.

### Boilerplate:

Class-based components require more boilerplate code to manage state and lifecycle methods, which can make the code more verbose and harder to read. Functional components with hooks have less boilerplate code, making them more concise and easier to read.

## The useState hook

**useState** is a React Hook that lets you add a state variable to your component.

## Parameters

1. **Intial State:** The **useState** hook in React takes an initial state as a parameter. The initial state can be a value of any type. If a function is passed as the initial state, it will be treated as an initializer function. The initializer function should be a pure function, taking no arguments, and returning a value of any type. React will call the initializer function when initializing the component and store the return value as the initial state.
   The initial state is only used during the first render and subsequent calls to useState with a new initial state will override the previous initial state.

## Returns

The **useState** hook returns an array with exactly two values:

1. The current state. During the first render, it will match the initialState you have passed.
2. The set function that lets you update the state to a different value and trigger a re-render.

**Example:** Usage useState hook

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

This code snippet uses the **useState** hook to define a state variable named **count** with an initial value of 0, and a function named **setCount** that can be used to update the state.
The set function returned by useState lets you update the state to a different value and trigger a re-render. You can pass the next state directly, or a function that calculates it from the previous state:

**Example 1:** Basic usage of state setter function

```
const [count, setCount] = useState(0);
setCount(1)
```

**Example 2:** Passing a callback function to set the state

```
const [count, setCount] = useState(0);
setCount((prevCount) => prevCount + 1);
```

**Note:** Using the state setter function with a callback allows you to access the latest state value at the time of the update and perform calculations or updates based on that value. This ensures that the state is always updated correctly and consistently, regardless of the timing of the updates.

## The useEffect hook

**useEffect** is a React Hook that lets you synchronize a component with an external system.

### Parameters

1. **Setup:** The **useEffect** hook in React takes a function as its first argument, which contains the logic of your effect. This function may also return a cleanup function. When your component is initially added to the DOM, React will execute the setup function you provided in the useEffect hook. On subsequent re-renders React will first call the cleanup function with the previous values. After that, React will run your setup function with the new values.

   Finally, when your component is removed from the DOM, React will execute your cleanup function one last time. This ensures that your component's side effects are properly added and removed throughout its lifecycle.

2. **Options (dependencies):** The **useEffect** hook in React takes a dependency array as its second argument, which contains all the reactive values referenced inside the setup code. These values include props, state, and all variables and functions declared directly in the component body. The list of

dependencies must have a constant number of items and be written inline in the form of **[dep1, dep2, dep3]**.

React compares each dependency with its previous value using a comparison algorithm. If you don't provide the dependency array, your effect will run after every re-render of the component.

## Returns

The **useEffect** hook does not return an value.

**Example 1:** Usage of useEffect hook

```
 useEffect(() => {
  setInterval(() => {
    setTimer((prev) => prev++);
  }, 1000);
}, []);
```

This code snippet uses the **useEffect** hook to set an interval that increments the state value of timer after every second. The effect runs only once on mount due to an empty dependency array.

**Example 2:** Usage of useEffect hook (with a cleanup function)

```
useEffect(() => {
  const interval = setInterval(() => {
    setTimer((prev) => prev++);
  }, 1000);

  return () => clearInterval(interval)
}, []);
```

This code snippet uses the **useEffect** hook to create a timer that updates every second. It sets up an interval to update the timer and clears it on unmount using the cleanup function. The effect runs only once on mount due to an empty dependency array.

## The useReducer hook

**useReducer** is a React Hook that lets you add a reducer to your component. It is typically used when you have complex state transitions that involve multiple sub-values or when the next state depends on the previous state.

It is a more powerful alternative to the useState hook and is particularly useful when managing state for large or deeply nested objects. The **useReducer** hook provides a simple API for dispatching actions and updating state in a predictable way.

### Parameters

1. **reducer:** In React, the **useReducer** hook takes a pure reducer function as its first argument, which defines how the state gets updated. The reducer function should take in the current state and an action as arguments and return the new state. The state and action can be of any type.

2. **initialState:** The value that represents the initial state of the component. This can be any value, including an object or an array.

### Returns

**useReducer** returns an array with exactly two values:

1. The current state. During the first render, it's set to the initialState.
2. The dispatch function that lets you update the state to a different value and trigger a re-render.

**Example:** Usage of useReducer hook

```
const [state, dispatch] = useReducer(reducer, initialState);
```

This code snippet uses the **useReducer** hook to define a state variable named **state** with an initial value of **initialState**, and a function named **dispatch** that can be used to dispatch updates to the state.

## The dispatch function

The **dispatch** function returned by **useReducer** lets you update the state to a different value and trigger a re-render. You need to pass the **action** as the only argument to the dispatch function

**Example:**

```
const [timer, dispatch] = useReducer(reducer, initialState)

const handleIncrement = () => {
  dispatch({ type: "INCREMENT_COUNT" });
};
```

This code snippet uses the **dispatch** function from the **useReducer** hook and passes an **action** object of type "INCREMENT_COUNT". The reducer function then checks this action type to update the state of the timer.

## Writing the reducer function

The **reducer** function used in useReducer hook of React is a pure function that takes the current state and an action as arguments, and returns the new state.

The reducer function evaluates the type of the action and updates the state based on the type of action.

**Example:**

```
const reducer = (state, action) => {
  switch (action.type) {
    case "incremented_age": {
      return {
        name: state.name,
        age: state.age + 1,
      };
    }
```

```
    case "changed_name": {
      return {
        name: action.nextName,
        age: state.age,
      };
    }
    default:
      return state;
  }
```

## Custom hooks

Custom hooks are functions in React that allow you to reuse stateful logic across multiple components. They follow the naming convention of starting with the word "use" and can be defined and used in the same way as the built-in hooks provided by React.

Custom hooks are a way to abstract and share logic that is not tied to a specific component, which makes your code more modular, easier to read and maintain. They can encapsulate complex stateful logic and make it easy to use in multiple components, without having to repeat the same code in each component.

Custom hooks can use other built-in hooks and can also be composed with other custom hooks, which makes it easy to create complex and reusable logic that can be used across different parts of your application.

**Example:** The **useLocalStorage** custom hook

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

const useLocalStorage = (key, initialValue) => {
  const [value, setValue] = useState(() => {
    const storedValue = localStorage.getItem(key);
    return storedValue !== null ? JSON.parse(storedValue) : initialValue;
  });
```

```
  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);

  return [value, setValue];
};
```

This custom hook allows you to store and retrieve data in the browser's localStorage. It takes a key and initial value as arguments and returns a state **value** and a **setValue** function to update it.

## Summary

We have discussed the advantages of using hooks in functional components instead of relying on class-based lifecycle methods. The three main hooks covered were **useState**, **useEffect**, and **useReducer**, with useState used for state management, useEffect for handling side effects, and useReducer for more complex state management. These hooks are a newer addition to React and make managing stateful logic in functional components easier.

## Summarising it

Let’s summarise what we have learned in this Lecture:

- What are hooks?
- Motivation to use hooks.
- Rules of hooks.
- Order of hooks.
- State in function and class based components.
- The useState hook and usage
- The useEffect hook and usage
- The useReducer hook and usage
- Custom hooks in React

## Some References:

- Lifecycle methods vs Hooks in React: link
- Hooks in React: link
- Custom hooks: link