# 1. Hooks in React!

## **1.1 The ‘useState()’**

useState is a React Hook that lets you add a state variable to your component[[1]](#footnote-1).

Hooks Usage in Functions: Hooks are exclusive to functions and cannot be used in class components.

Execution Order: Hooks like useState() are executed in the order they appear in your function.

    useState()

    useState()

    useState()

    useState()

Avoid Conditional Placement: Hooks, such as useState(), should not be placed within conditions or loops.

//incorrect

if (isTrue) {

        useState()

    }

*Error: React Hook "useState" is called conditionally. React Hooks must be called in the exact same order in every component render  react-hooks/rules-of-hooks*

### 1.1.1 Using useState()

Simple Implementation:

Call the useState() function with a default value, like useState(4).

The default value is passed in parentheses.

Returns an array, which we destructure as below into count (current state) and setCount (state update function).

*const* [count, setCount] = useState();

Avoid Overriding Issues:

Problematic Counter Implementation:

*function* incrementCount() {

    setCount(count + 1);

    setCount(count + 1);

  }

Issue 1:

* Overriding Values: The counter is incremented twice in rapid succession, but since setCount is synchronous, the second setCount call overrides the value set by the first one. As a result, both updates yield the same value.

Issue 2:

* Full Page Re-render: Every time setCount is called, it triggers a re-render of the entire component. This can be resource-intensive, especially if there are complex tasks or calculations involved in rendering the page.

Solution? Using a Function

Use an arrow or a normal function inside the setCount()

*function* incrementCounter() {

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

  }

*function* decrementCounter() {

    setCount((*prevCount*) *=>* {

      return *prevCount* - 1;

    });

  }

Solution: Using a Function Parameter: Using a function as an argument to setCount, React ensures that the state update is based on the previous state (prevCount). This prevents overriding issues.

### 1.1.2 Complexity in useState()

Complex Default State: If the default state involves complex logic, rendering it directly for every click can be inefficient.

To avoid this, we use a function inside the useState()

//if default state  is passed directly it gets rendered for every click

*function* defaultState() {

  console.log("This is running");

  return 5;

}

Right Way

*const* [count, setCount] = useState(() *=>* {

    return defaultState();

  });

### 1.1.3 Multiple useState() hooks

Simplify State Management: Using multiple useState() hooks makes state management more modular.

 // //using multiple hooks

*const* [count, setCount] = useState(() *=>* defaultState());

*const* [theme, setTheme] = useState("");

Updating Multiple States: Update multiple states independently, preventing conflicts.

*function* incrementCounter() {

    setCount((*prevCountValue*) *=>* {

      return *prevCountValue* + 1;

    });

    setTheme("Green");

  }

*function* decrementCounter() {

    setCount((*prevCount*) *=>* *prevCount* - 1);

    setTheme("red");

  }

## **1.2 The ‘useEffect()’**

### 1.2.1 What is useEffect()

A side effect when something happens.

Structure of useEffect()

useEffect(() *=>*{

  //The code that we want to run

  //Optional return function

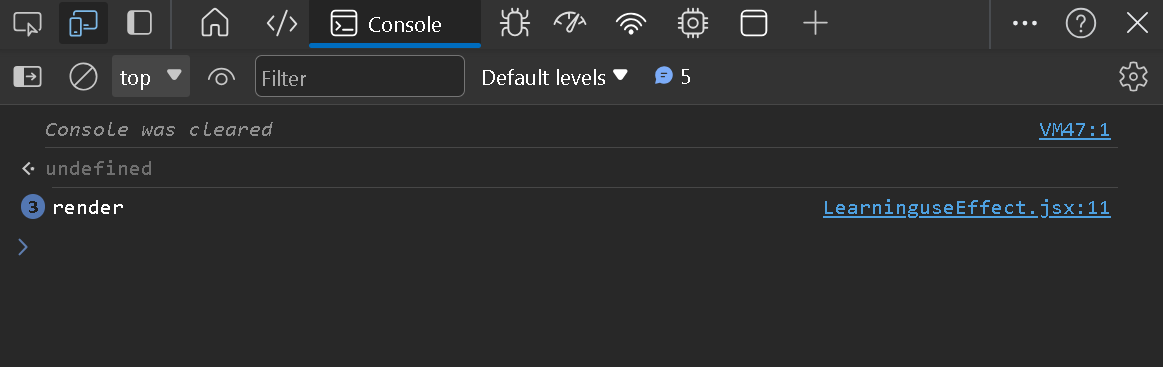
  }, []) // dependency array

useEffect(() *=>* {

    console.log("render");

  });

Output:



But if we keep clicking the same it will not render.

### 1.2.2 useEffect() – Parameters

[] – the dependency array

  useEffect(() *=>* {

    console.log(`Resource type changed ${resource}`);

  }, [resource]);

This ‘hook’ will only be executed when something in the parameters, [resource], changes. These are what something has to happen whenever something changes.

### 1.2.3 On Mount Effect

  useEffect(() *=>* {

    console.log(`Resource type changed ${resource}`);

  }, []);

Since no values are changing this will only get executed at the start and never get rendered.

This can also be used as a checking effect that checks and changes whenever a change happens.

  //Window width and height calculation

*const* [width, setwidth] = useState(window.innerWidth);

*const* [height, setHeight] = useState(window.innerHeight);

  //width

*const* widthHandler = () *=>* {

    setwidth(window.innerWidth);

  };

  useEffect(() *=>* {

    window.addEventListener("resize", widthHandler);

  }, []);

  //height

*const* heightHandler = () *=>* {

    setHeight(window.innerHeight);

  };

  useEffect(() *=>* {

    window.addEventListener("resize", heightHandler);

  });

  return (

    <div className="state-common-style use-effect">

      <h2>Learning useEffect()</h2>

      <div className="dimensions">

        <h3>

          Window Width: <span>{width}</span>

        </h3>

        <h3>

          Window Height: <span>{height}</span>

        </h3>

      </div>

### 1.2.4 The Side Effect

  useEffect(() *=>* {

    //Fetching data from JSON

    fetch(`https://jsonplaceholder.typicode.com/${resource}`)

      .then((*response*) *=>* *response*.json())

      .then((*json*) *=>* console.log(*json*));

  }, [resource]);

Whenever we are changing the ‘resource’ we are running a side effect of our code, so that gets printed out in the console when a ‘resource’ changes.

## **1.3 The ‘useMemo()’**

1. https://react.dev/reference/react/useState [↑](#footnote-ref-1)