Software Studio
軟體設計與實驗

Web Application Framework (Part II)

Hung-Kuo Chu

Department of Computer Science

National Tsing Hua University

CS2410

Outline

- Functional Component
- Introduction to React Hook
- State Hook
- Effect Hook
- Rules of Hook
- Custom Hook

Class Component

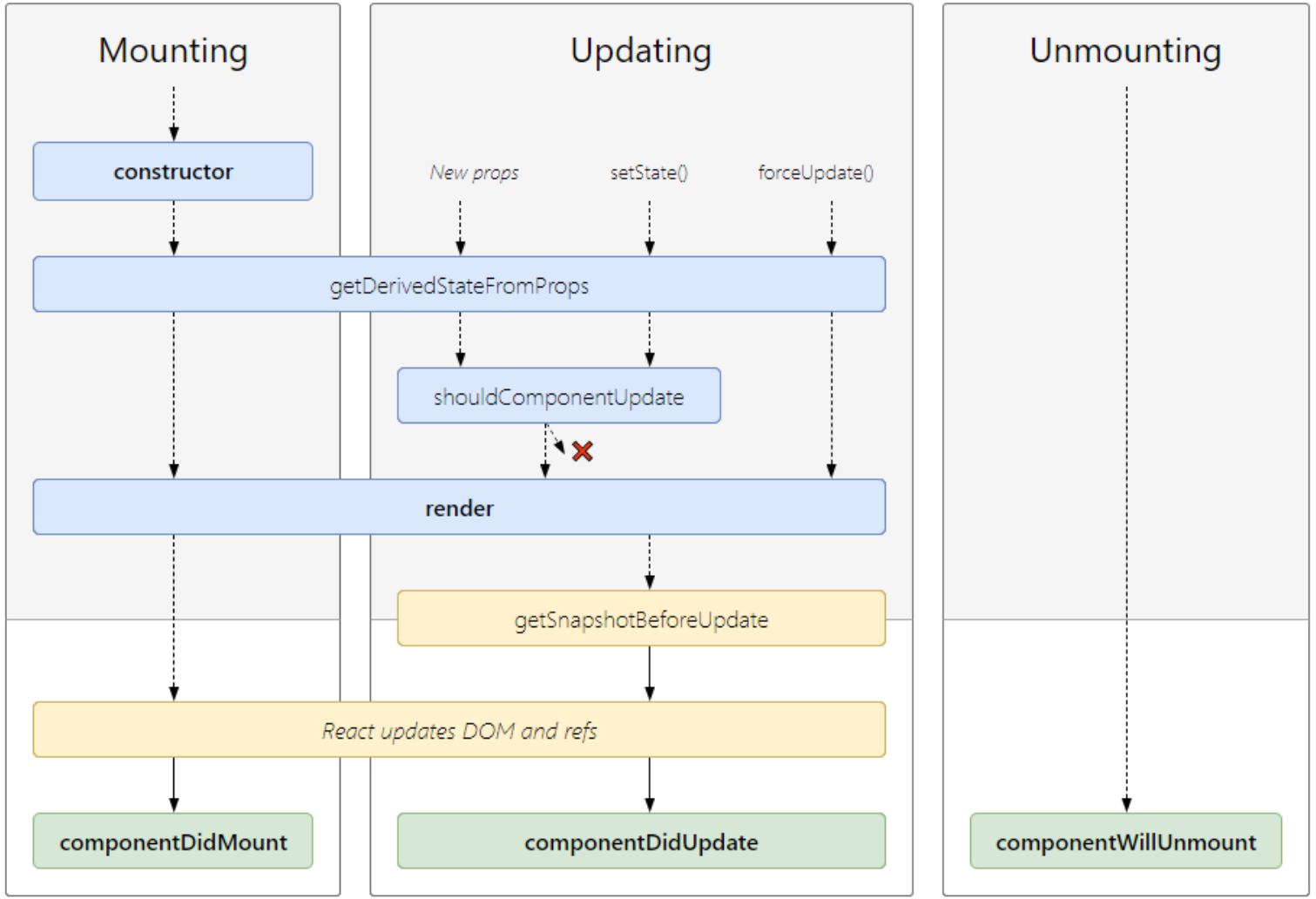
- Object that extends React.Component
- Must implement a **render()** method that returns a React element
- Stateful component with logic and state
  - Initialize state in the constructor
- Implement React lifecycle methods, e.g., componentDidMount().

Class Component Example

```
import React from ‘react’;

class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
  render() {
    return (
      <div>
        <button onClick={() => this.setState({count: this.state.count+1})}>
          Click
        </button>
      </div>
    );
  }
}
```

Class Component Lifecycle



React.Component

Functional Component

- Plain JS function that accepts **props** as argument and returns a **React element**
- **Render()** method is not required
- **Stateless** components
- Cannot implement React lifecycle methods
- Using **Hooks** to enable React state and lifecycle features.

Functional Component Example

- The simplest way to define a component in React is to write a JS function

```
function Welcome (props) {
  return (
    <hi>Hello, {props.name}</hi>
  );
}
```

Why Functional Component?

- Easier to read, test and debug
  - don’t worry about hidden states and side effect
  - plain JavaScript function
- Better performance
  - less code, faster bundles

Ref: https://ithelp.ithome.com.tw/articles/10234746

Design Guideline

- **Class Component** is preferred when we need an **internal state** to control the behavior of the React component

- **Functional Component** is preferred when the component can be rendered only based on the **props**

React Hook

- What do we do when we need a functional component with **state** and **lifecycle features?**

Introduction to Hook

- Functions let you **hook into** React state and lifecycle features
- Allow you to use React without declaring classes
- React provides built-in Hooks like **useState**
- Create custom Hooks to reuse stateful behavior between components

Functional Component Lifecycle

"Render phase"

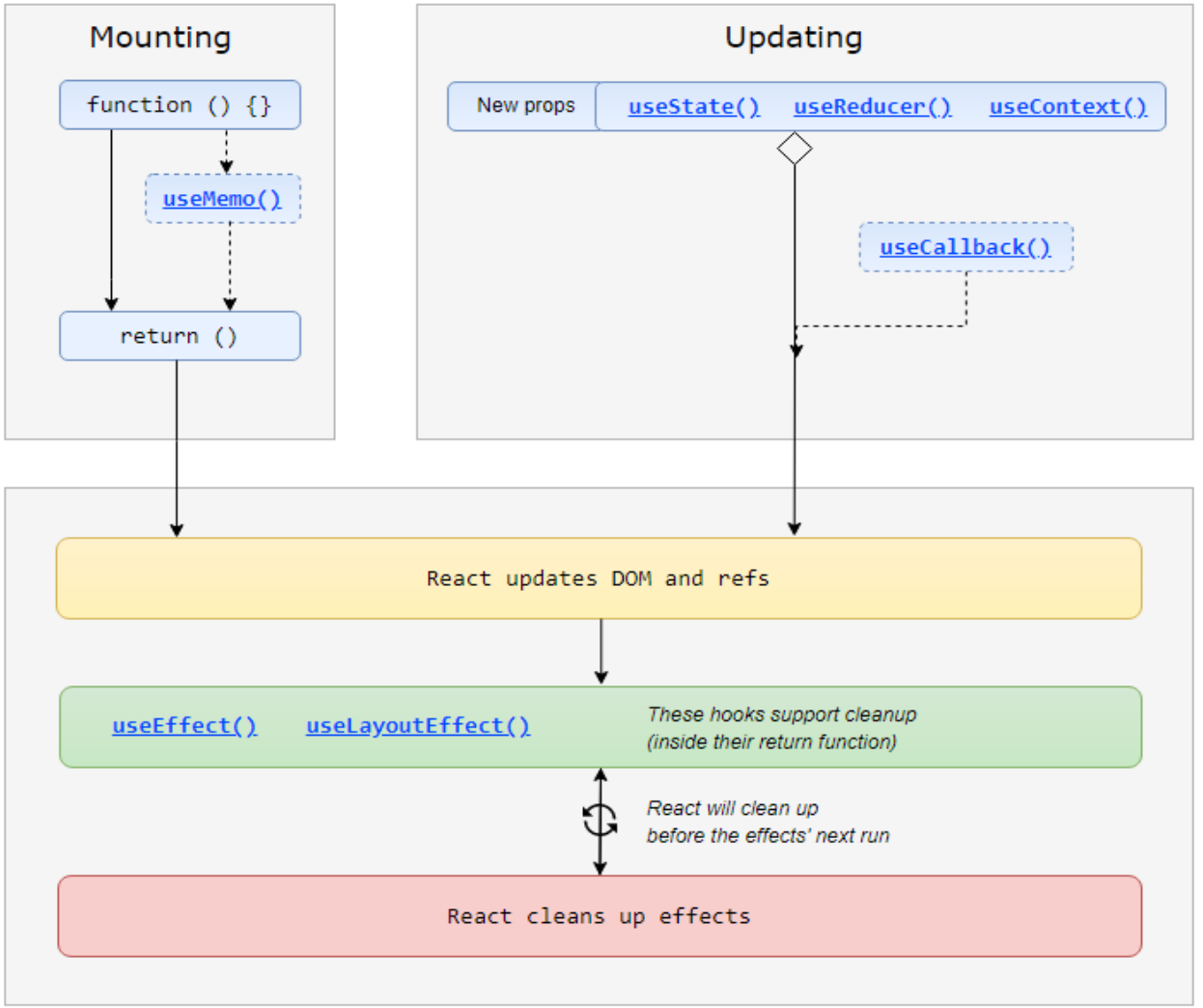
Pure and has no side effects. May be paused, aborted or restarted by React.

"Commit phase"

Can work with DOM, run side effects, schedule updates.

"Cleanup phase"

Runs before a component is removed. Prevents memory leaks.



Wavez/react-hooks-lifecycle

Made with ❤ by Gal Margalit

State Hook

"Render phase"

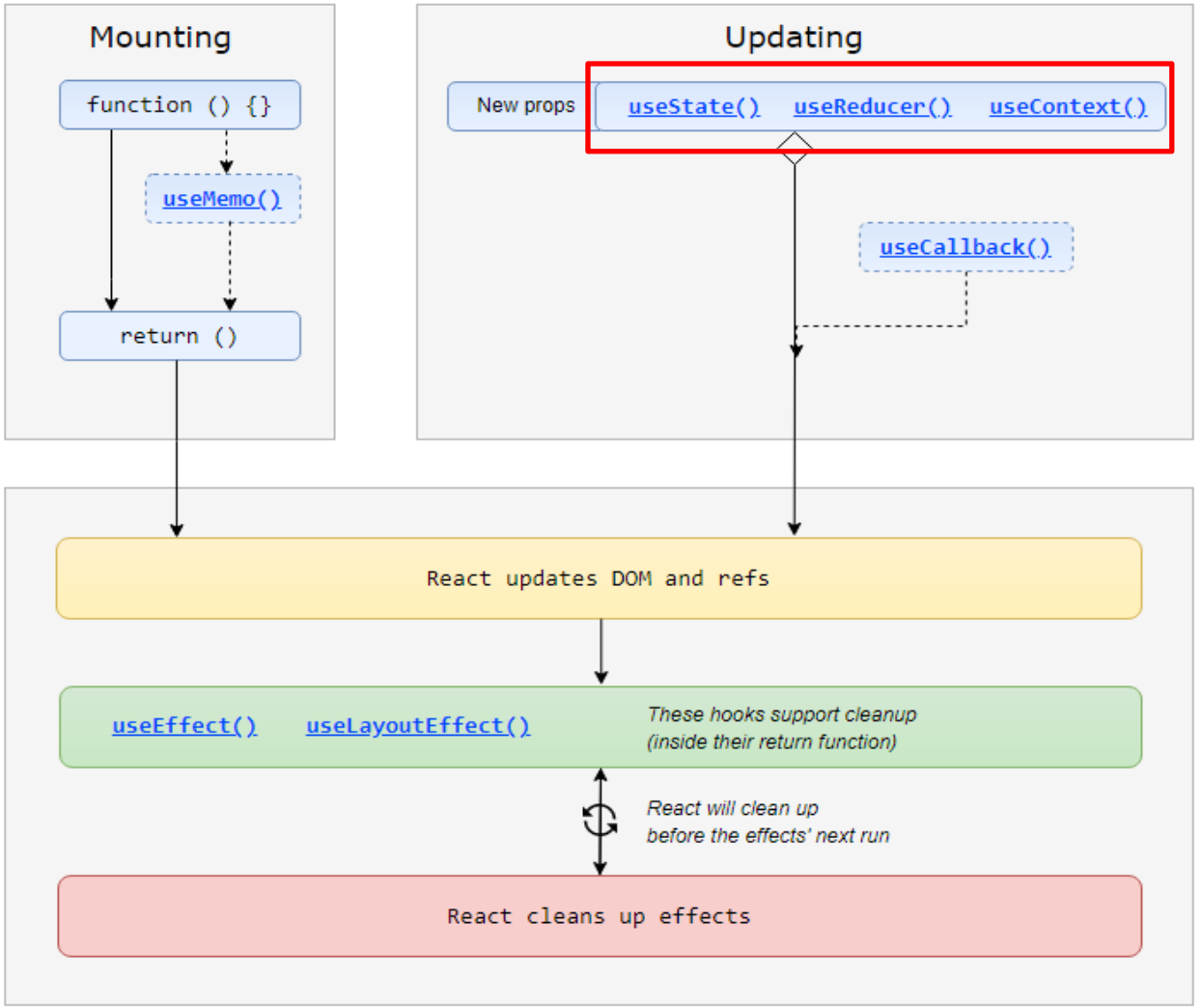
Pure and has no side effects. May be paused, aborted or restarted by React.

"Commit phase"

Can work with DOM, run side effects, schedule updates.

"Cleanup phase"

Runs before a component is removed. Prevents memory leaks.



Wavez/react-hooks-lifecycle

Made with ❤ by Gal Margalit

useState

- To use the same capabilities of **this.state** provided in a class component
- Normal variables disappear when function exits, but state variables are preserved by React

Class Component Example

```
import React from ‘react’;

class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
  render() {
    return (
      <div>
        <button onClick={() => this.setState({count: this.state.count+1})}>
          Click
        </button>
      </div>
    );
  }
}
```

Functional Comp. Counterpart

```
import React, { useState } from 'react';

function Example (props) {
  const [ count, setCount ] = useState(0);

  return (
    <div>
      <button onClick={() => setCount(count + 1)}>
        Click
      </button>
    </div>
  );
}
```

useState(): Argument

- The only argument passed to **useState** is the initial value of the state variable

```
import React, { useState } from 'react';

function Example (props) {
  // Declare a new state variable count with initial value 0
  const [ count, setCount ] = useState(0);
  …
```

useState(): Return

- **useState** returns a pair of values
  - The current state
  - A function that updates it

```
import React, { useState } from ‘react’;

function Example (props) {
  // count: the current state
  // setCount: function that updates count
  const [ count, setCount ] = useState(0);
  …
```

Array Destructuring

Reading State

- In a class component

```
<p>You clicked { this.state.count } times</p>
```

- In a functional component

```
<p>You clicked { count } times</p>
```

Updating State

- In a class component

```
<button onClick={ () => this.setState({ count: this.state.count + 1 })
}>
  Click
</button>
```

- In a function component

```
<button onClick={ () => setCount(count + 1) }>
  Click
</button>
```

Effect Hook

"Render phase"

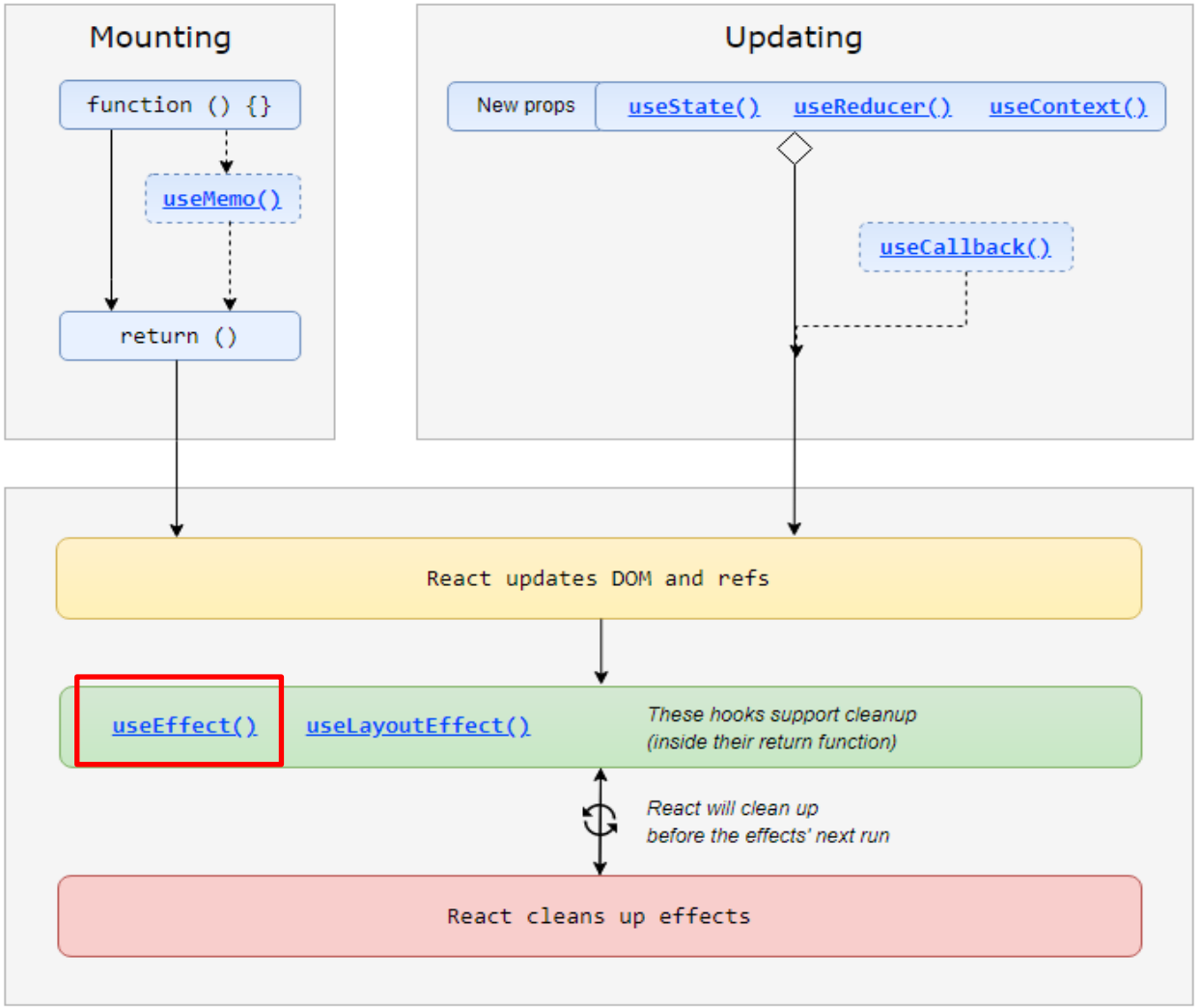
Pure and has no side effects. May be paused, aborted or restarted by React.

"Commit phase"

Can work with DOM, run side effects, schedule updates.

"Cleanup phase"

Runs before a component is removed. Prevents memory leaks.



Effect Hook

- **Side effects**: operations that affect other components and can't be done during rendering
  - Operations **after DOM updates**, e.g., fetch data, subscription, changing DOM from React components.
- **useEffect**: perform side effects from a function component, like:
  - **componentDidMount (Mounting)**
  - **componentDidUpdate (Updating)**
  - **componentWillUnmount (Unmounting)**

useEffect(): Example

```
import React, { useState, useEffect } from 'react';

function Example (props) {
  const [ count, setCount ] = useState(0);
  // Similar to componentDidMount and componentDidUpdate
  useEffect(() => {
    // Update title after every rendering
    document.title = `You clicked ${ count } times`;
  })
  return (
    <div>
      <button onClick={() => setCount(count + 1)}>
        Click
      </button>
    </div>
  );
}
```

useEffect(): Example

- In a class component

```
componentDidMount() {
  document.title = `You clicked ${ this.state.count } times`;
}

componentDidUpdate() {
  document.title = `You clicked ${ this.statecount } times`;
}
```

- In a function component

```
useEffect(() => {
  document.title = `You clicked ${ count } times`;
})
```

Cleans Up Effects

"Render phase"

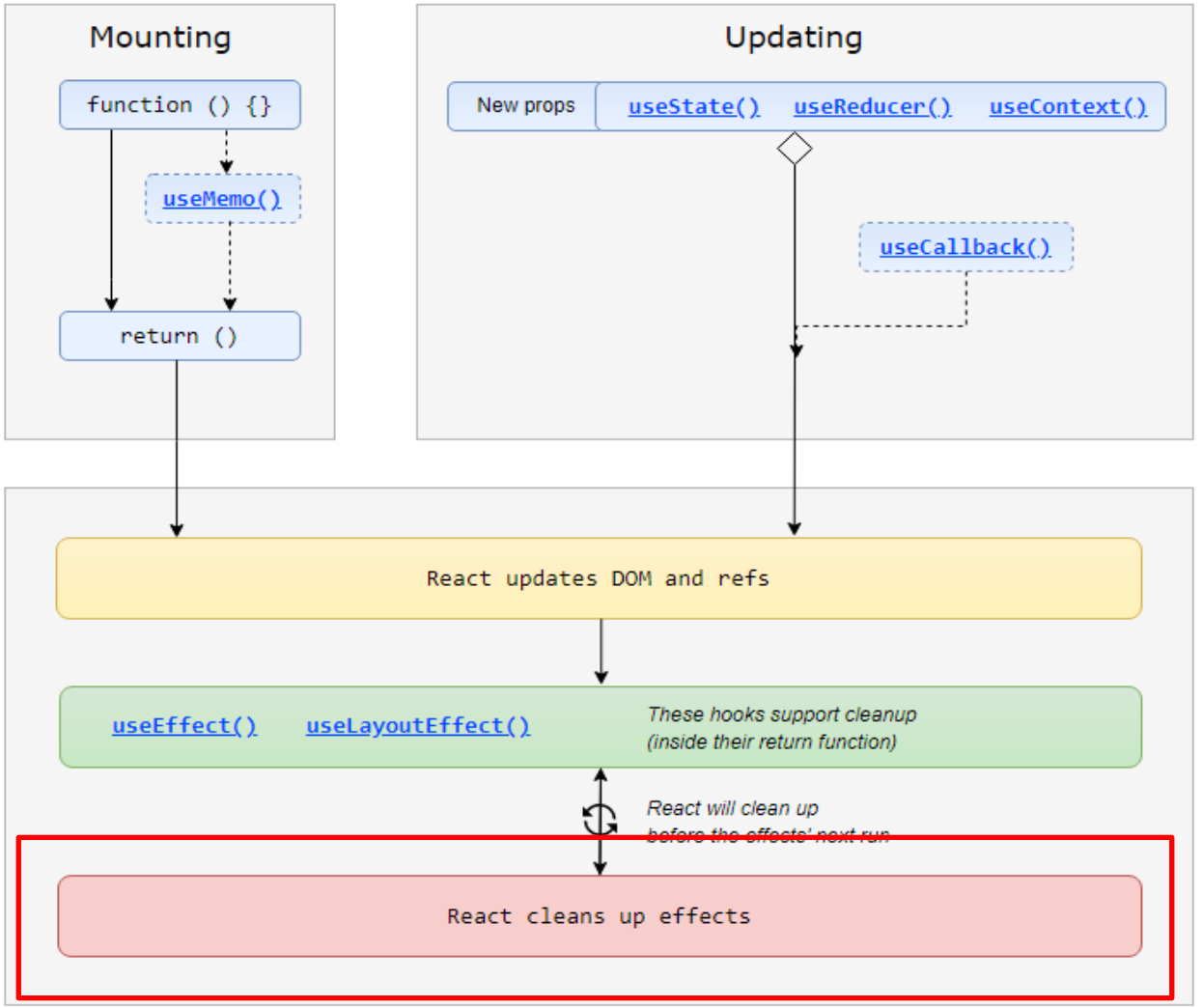
Pure and has no side effects. May be paused, aborted or restarted by React.

"Commit phase"

Can work with DOM, run side effects, schedule updates.

"Cleanup phase"

Runs before a component is removed. Prevents memory leaks.



Wavez/react-hooks-lifecycle

Made with ❤ by Gal Margalit

Cleans Up Effects

- **useEffect()** returns **a cleanup function** to helps developer clean effects that prevent unwanted behaviors and optimizes application performance.
- **When**
  - Component is going to unmount
  - Before the execution of the next scheduled effect

Understanding React's useEffect cleanup function

Example

```
function FriendStatus (props) {
  const [ isOnline, setIsOnline ] = useState(null);

  useEffect(() => {
    // Subscribe to friend status
    ChatAPI.subscribeFriendStatus(props.id, (isOnline) => {
      setIsOnline(isOnline);});

    // Return a cleanup function to specify how to cleanup after effect
    return function cleanup() {
      ChatAPI.unsubscribeFriendStatus(props.id, (isOnline) => {
        setIsOnline(isOnline);})
    };
  })
  …
```

useEffect(): Dependency

- useEffect accepts 2 arguments
  - **callback:** function contains the side effect logic
  - **dependencies:** an optional array of dependencies, useEffect() executes callback only if the dependencies have changed between renderings

```
useEffect(callback, dependencies);
```

useEffect(): Dependency

- after every render

```
useEffect(() => {
  // Runs after every rendering
  …
});
```

- once

```
useEffect(() => {
  // Runs ONCE after initial rendering
  …
}, [ ]);
```

useEffect(): Dependency

- on state change

```
const [ state, setState ] = useState();

useEffect(() => {
  // Runs when state changes
  …
}, [ state ]);
```

- on states change

```
const [ counter1, setCounter1 ] = useState(0);
const [ counter2, setCounter2 ] = useState(0);

useEffect(() => {
  // Runs when counter1 or counter2 changes
  …
}, [ counter1, counter2 ]);
```

useEffect(): Dependency

- on prop change

```
const { prop } = props;

useEffect(() => {
  // Runs when prop changes
  …
}, [ prop ]);
```

- on props change

```
const { id1, id2 } = props;

useEffect(() => {
  // Runs when id1 or id2 changes
  …
}, [ id1, id2 ]);
```

useEffect(): Dependency

- on unmount

```
const [ state, setState ] = useState();

useEffect(() => {
  return () => {
    // cleanup function here
    …
  }
}, [ state ]);
```

Rules of Hook

- Only call Hooks at the top level
- Don’t call Hooks inside
  - Loops
  - Conditions
  - Nested functions
- You can call Hooks from
  - React function components
  - Custom Hooks

Example

```
function Form () {
  // 1. Use the id state variable
  const [ userId, setUserId ] = useState(0);

  // 2. Use an effect for fetching data
  useEffect(function fetchData() {
    storageAPI.fetchData(userId);
  });

  // 3. Use the name state variable
  const [ userName, setUserName ] = useState('James');

  // 4. Use an effect for updating the title
  useEffect(function updateTitle() {
    document.title = userName + ' s Form';
  });
}
```

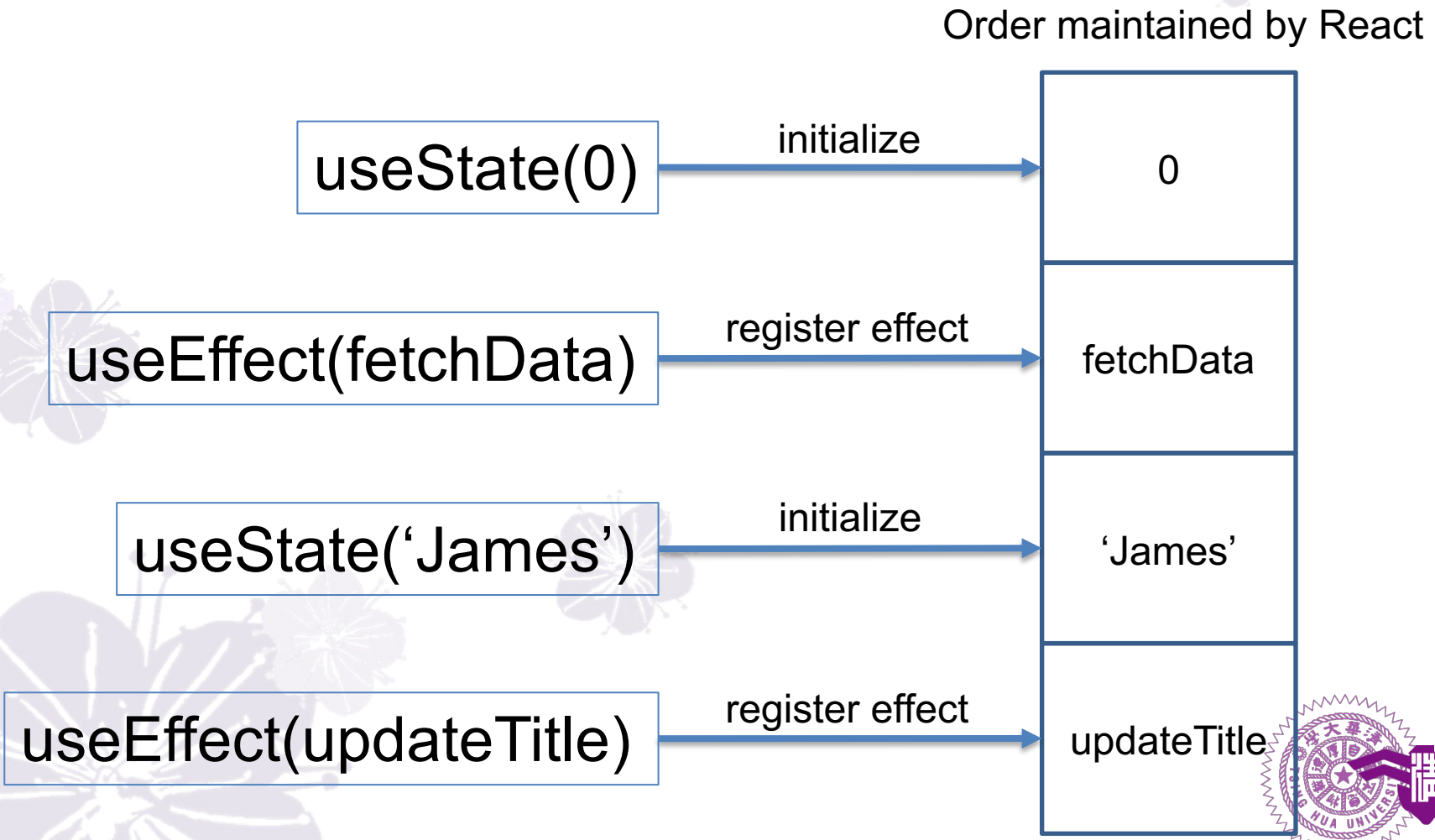
Rules of Hook

- How does React know which state corresponds to which **useState** call?
- React relies on the order in which Hooks are called

```
// First render
useState(0);                    // 1. Initialize the id state variable
useEffect(fetchData);           // 2. Add an effect for fetching data
useState('James');              // 3. Initialize the name state variable
useEffect(updateTitle);         // 4. Add an effect for updating the title
// Second render
useState(0);                    // 1. Read the id state variable
useEffect(fetchData);           // 2. Replace effect for fetching data
useState('James');              // 3. Read the name state variable
useEffect(updateTitle);         // 4. Replace effect for updating the title
```

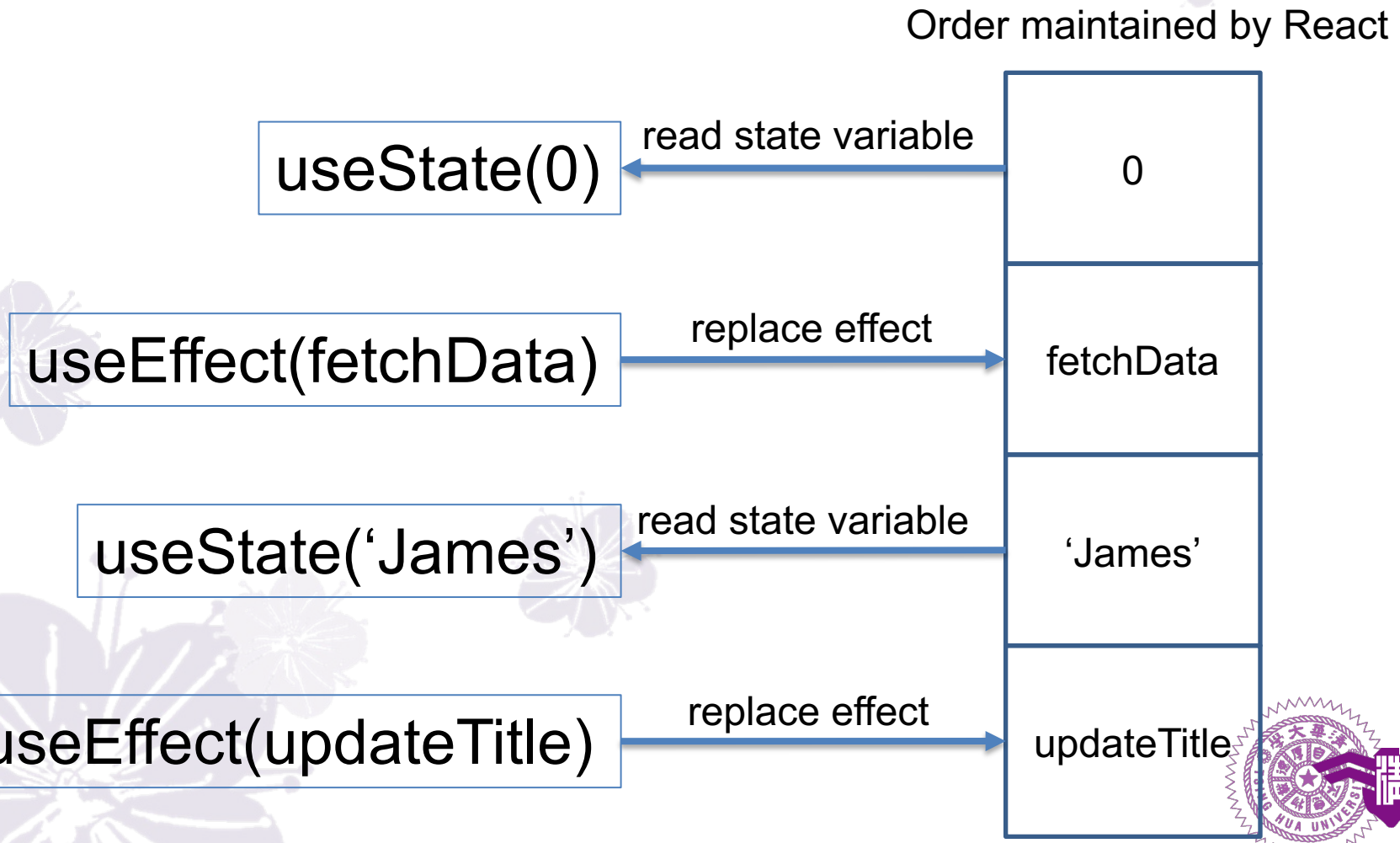
Rules of Hook

- Executes **Form(): 1st** time



Rules of Hook

- Executes **Form(): 2nd** time



Rules of Hook

- What happens if we put a Hook call inside a condition?

```
if (userId != null) {
  // Breaking the first rule by using a Hook in a condition
  useEffect(function fetchData() {
    storageAPI.fetchData(userId);
  })
}
```

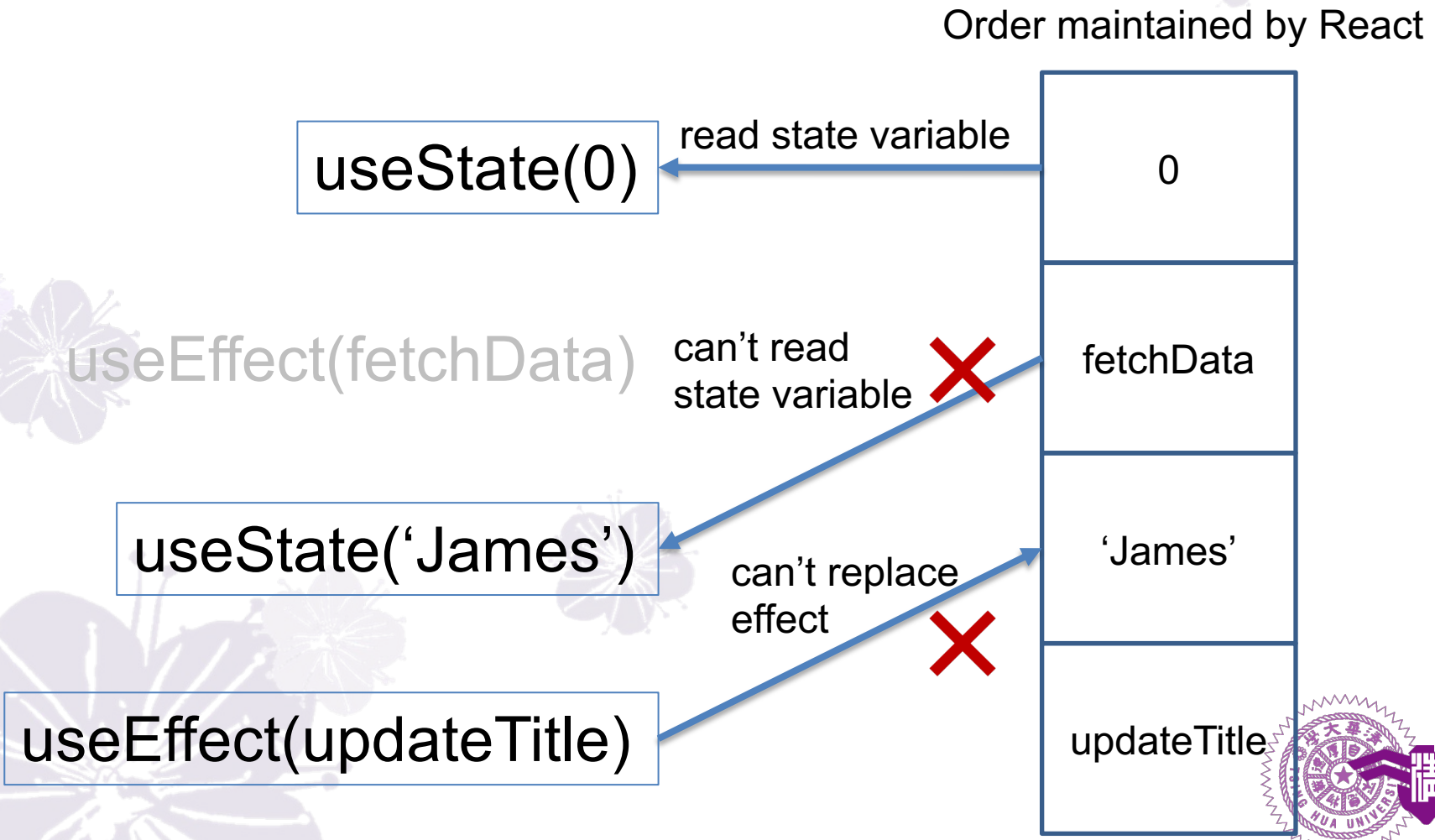
Rules of Hook

- **userId** might become null if the user clear the form, making the condition false
- The order of the Hook calls becomes different

```
useState(0)                    // 1. Read the id state variable
// useEffect(fetchData)        // Skipped
useState('James')              // 2 (but was 3). Fail to read name
useEffect(updateTitle)         // 3 (but was 4). Fail to replace effect
```

Rules of Hook

- Execute **Form()** will skip the fetchData



Rules of Hook

- React expects that the second Hook call corresponds to the **fetchData** effect
- Wouldn't know what to return for the second **useState** Hook call

```
useState(0)                    // 1. Read the id state variable
// useEffect(fetchData)        // Skipped
useState('James')              // 2 (but was 3). Fail to read name
useEffect(updateTitle)         // 3 (but was 4). Fail to replace effect
```

Rules of Hook

- If we want to run an effect conditionally, we can put that condition inside our Hook

```
useEffect(function fetchData() {
  // Not breaking the first rule
  if (userId != null) {
    storageAPI.fetchData(userId);
  }
})
```

Custom Hook

- Sometimes we want to share the same stateful logic between components
- We can build our own Hooks to extract component logic into reusable functions

Example

```
function FriendStatus (props) {
  const [ isOnline, setIsOnline ] = useState(null);

  useEffect(() => {
    // Subscribe to friend status
    ChatAPI.subscribeFriendStatus(props.id, (isOnline) => {
      setIsOnline(isOnline);
    });
    // return a cleanup function
    return function cleanup() {
      ChatAPI.unsubscribeFriendStatus(props.id, (isOnline) => {
        setIsOnline(isOnline);
      })
    };
  })
  …
```

1. Copy and paste?
2. Duplicate codes
3. Difficult to maintain

```
function FriendListItem (props) {
  // Also wants to subscribe to friend status


  …
```

Custom Hook: Definition

- Extract our logic to another function
- Function name always starts with **use**

```
function useFriendStatus (friendID) {
  const [ isOnline, setIsOnline ] = useState(null);

  useEffect(() => {
    // Subscribe to friend status
    ChatAPI.subscribeFriendStatus(friendID, (isOnline) => {
      setIsOnline(isOnline);
    });
  return function cleanup() {
      ChatAPI.unsubscribeFriendStatus(friendID, (isOnline) => {
        setIsOnline(isOnline);
      })
    };
  })

  return isOnline;
}
```

Custom Hook

- Use our logic in different components
- State and effects are isolated

```
function FriendStatus (props) {
  const isOnline = useFriendStatus(props.id);

  return isOnline ? ‘Online’ : ‘Offline’;
}
```

```
function FriendListItem (props) {
  const isOnline = useFriendStatus(props.id);

  return (
    <li style={{ color: isOnline ? ‘green’ : ‘red’ }}>
      {props.name}
    </li>
  );
}
```


thank you!
Question?