React Hooks

slides at https://github.com/mvolkmann/talks

R. Mark Volkmann
Object Computing, Inc.
http://objectcomputing.com
Email: mark@objectcompuing.com
Twitter: @mark_volkmann
GitHub: mvolkmann
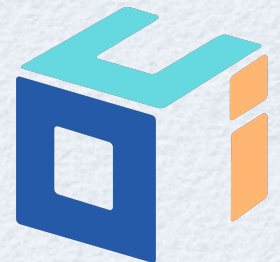
OCI | TRAINING

Hooks Overview

- Added in React version **16.8** released February 2019

- **Enables implementing stateful components
with functions instead of classes**

- No plans to remove existing ways of implementing components

- Hooks provide new way that
most developers will find easier to understand

- Components that use hooks can be used
together with class-based components

- Existing apps can gradually incorporate hooks
or never use them … you really should!

- Makes React the only major web framework that supports
implementing an entire app with only functions

  - Svelte can do this too

2

Hook Caveats

- There are some lifecycle methods whose functionality cannot yet be implemented using hooks
  - `componentDidCatch`
  - `getSnapshotBeforeUpdate`
- These are rarely used

Hook Benefits

- Implementing components with functions removes need to

  - understand `this` keyword

  - understand `bind` and when to use it

- Easier to work with component state and context

- Easier to reuse state logic between multiple components

- Removes most needs for higher-order components and render props

  - these require increased levels of code nesting

- Uses "effects" in place of lifecycle methods

  - makes it possible to better organize related code such as adding/removing event listeners and opening/closing resources

- Easier to optimize function component code than class component code

  - refers to minifying, hot reloading, and tree shaking

Hook Function Rules

- Names should start with **`use`**

  - allows linting rules to check for proper use of hooks

  - provides clue that the function may access state

- Can only be called in
function-based components
and in custom hooks

- Cannot be called conditionally

  - means they cannot be called in
  `if/else` blocks, loops, or nested functions

  - ensures that for any component,
  the **same hooks are invoked in
  the same order on every render**

ESLint

- React provides ESLint rules to detect hook rule violations

    - see https://www.npmjs.com/package/eslint-plugin-react-hooks

    - should configure all with value "`error`"

- Rule "`react-hooks/rules-of-hooks`"

    - assumes any function whose name begins with "`use`"
      followed by an uppercase letter is a hook

    - verifies hooks are only called from
      function components (name starts uppercase)
      or custom hook functions

    - verifies that hooks will be called
      in the same order on every render

- Rule "`react-hooks/exhaustive-deps`"

    - verifies the list of dependencies for hooks like `useEffect`
      based on data used by the hook

6

Provided Hooks

- Implemented as functions exported by `react` package
  - ex. `import React, {useEffect, useState} from 'react';`
- Described on following slides,
  somewhat ordered based on
  how frequently they are used

7

State Hook …

- **useState** provides way to add state to function components

  - allows components to use state without using **this** keyword

  - takes initial value of state

  - returns array containing current value and a function to change it

- The following can appear inside a function that defines a component

```
const [petName, setPetName] = useState('Dasher');
const [petBreed, setPetBreed] = useState('Whippet');
```

  - **petName** holds current value of state

  - **setPetName** is a function that can be called to change value

- "set" functions

  - can be passed a new value OR
    a function that will be passed current value and returns new value

  - **calls to them trigger the component to be re-rendered
    if the value changes**

8

... State Hook ...

- State is often a primitive value, but can also be an object or array (also an object)

- When an object is used

  - calls to corresponding set function must pass entire value

  - set functions do not merge top level properties
    of the object passed to them with the current value
    as is done by the `Component setState` method

- `useState` calls are made every time the component is rendered

  - to obtain current value for each piece of state

  - initial values are only be applied in first render

... State Hook ...

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

export default function Pet() {
  const [petName, setPetName] = useState('Dasher');
  const [petBreed, setPetBreed] = useState('Whippet');
  const changeBreed = e => setPetBreed(e.target.value);
  const changeName = e => setPetName(e.target.value);
  return (
    <div>
      <label>
        Name
        <input onChange={changeName} value={petName} />
      </label>
      <br />
      <label>
        Breed
        <select onChange={changeBreed} value={petBreed}>
          <option>Greyhound</option>
          <option>Italian Greyhound</option>
          <option>Whippet</option>
        </select>
      </label>
      <div>{petName} is a {petBreed}.</div>
    </div>
  );
}
```

React Hooks

... State Hook

- Not necessary to understand how this works,
  but it's interesting

- State values for a component are stored in a **linked list**

- Each call to `useState` associates a state value
  with a different node in the linked list

- In previous example

  - `petName` is stored in first node

  - `petBreed` is stored in second node

top-state-hook

- But what if you wanted to share state between multiple components?

- Custom hook **useTopState** adds state outside components

    - takes state name (any string) and initial value

    - returns current value and a function for changing it, just like **useState**

```
const [petName, setPetName] = useTopState('petName', 'Dasher');
```

- Can be shared with any number of other components

    - only first component to use a given state name initializes it

- If any component changes the state,
  all components that use it are re-rendered

- See https://www.npmjs.com/package/top-state-hook

Effect Hook ...

- **useEffect** provides an alternative to some lifecycle methods in function components
  - **componentDidMount**
  - **componentDidUpdate**
  - **componentWillUnmount**

- Effects have two phases, setup and cleanup
  - think of setup as being performed when a class component would call **componentDidMount** or **componentDidUpdate**, which is after React updates the DOM
  - think of cleanup as being performed when a class component would call **componentWillUnmount**

- Example use
  - move focus to a particular input - demonstrated in "Ref Hook" section later

… Effect Hook …

- Setup functionality examples
  - register an event listener
  - open a network connection
  - start a timeout or interval
  - fetch data by calling a REST service

- Cleanup functionality examples
  - unregister the event listener
  - close the network connection
  - clear the timeout or interval

... Effect Hook ...

- Function passed to **useEffect** performs setup

- If no cleanup is needed, this function returns nothing

- If cleanup is needed, this function returns another that performs cleanup

- Example

```
useEffect(() => {
  console.log('performing setup');
  return () => {
    console.log('performing cleanup');
  };
});
```

- **useEffect** can be called any number of times
  inside a function component

- Typically called once for each distinct kind of effect
  rather than combining code for multiple effects in a single call

15

... Effect Hook ...

- **In first render** of a component, order of execution is

  1. all code in component function

  2. setup code in all effects in order defined

- **In subsequent renders**, order of execution is

  1. all code in component function

  2. cleanup code in all effects in order defined (not reverse)

  3. setup code in all effects in order defined

... Effect Hook

- **To prevent cleanup/setup code from running in every subsequent render**

  - supply second argument to `useEffect` that is an array of variables

  - cleanup/setup code is only run again if any of these variable values have changed since last call

  - to only run on first render, pass <u>empty array</u>

- **Forgetting** to pass second argument causes effect to run on **every render**

  - ESLint rule "`react-hooks/exhaustive-deps`" warns if setup function uses values defined outside it

- We will see examples later

Context Hook ...

- **useContext** provides an alternative way to consume context state in function components

- Does not change how context providers are implemented

  - still implemented by creating a class that extends from **React.Component** and renders a **Provider**

  - for details, see https://reactjs.org/docs/context.html

- Example

  - suppose a context provider has been implemented in component **SomeContext**

  - **useContext** can be used in another component to access its state

```
import {SomeContext} from './some-context';

export default function MyComponent() {
  const context = useContext(SomeContext);
  return <div>{context.someData}</div>;
}
```

... Context Hook

- **`context`** variable is set to an object that provides
  - read-only access to state properties of the context
  - ability to call methods defined on it
- Directly setting properties on **`context`**
  affects the local object, but not context state
  - not flagged as an error
- Context methods can provide a way for
  context consumers to modify context state
- Calling **`useContext`** also
  subscribes the component to context state updates
  - whenever context state changes, the component is re-rendered
- To avoid re-rendering the component on every context state change,
  wrap returned JSX in a call to **`useCallback`** described soon

19

context-easy

- A great use of **useContext** is in conjunction with the npm package "context-easy"

  - implements a Context API **Provider** that can manage all the state in an application, like a Redux store

  - highly generic, making it suitable for any application

  - can get and set state data at dot-separated paths

    - ex. **user.address.city**

  - provides context methods **get**, **set**, **transform**, **delete**, **push**, **pop**, **map**, **filter**, **increment**, **decrement**, and **toggle**

  - provides **Input**, **TextArea**, **Select**, **Checkboxes**, and **RadioButtons** components that tie underlying form elements to state paths

  - saves data in **sessionStorage** and restores from there so browser refresh doesn't UI state

- See https://www.npmjs.com/package/context-easy

20

Callback Hook ...

- **useCallback** takes an expression and
  an array of variables that affect the result

- Returns a memoized value

- Often the input expression is a function

- Can be used to avoid recreating callback functions
  defined in function components
  every time they are rendered

  - but the functions they call are still recreated on each render!

  - more about this on next slide

- Such functions are often used as DOM event handlers

... Callback Hook ...

- For example, consider difference between these

```
<input onChange={e => processInput(e, color, size)}>

<input onChange={useCallback(e => processInput(e, color, size), [color, size]}>
```

- Same functionality, but second line
  only creates a new function for `onChange` prop ...

  > But the arrow function
  > passed to `useCallback`
  > IS recreated on every render.

  - on first render

  - on subsequent renders only if value of `color` or `size` has changed since last render

- Avoiding creation of new callback functions
  allows the React reconciliation process to correctly
  determine whether the component needs to be re-rendered

- Provides a performance benefit

... Callback Hook ...

- If callback function does not depend on any variables,
  pass an empty array for second argument

  - causes **useCallback** to always return same function

- If second argument is omitted,
  a new function will be returned on every call
  which defeats the purpose

Why doesn't **useCallback** treat omitting the second argument the same as passing an empty array?

… Callback Hook

- **useCallback** can also serve as a substitute
  for the lifecycle method **shouldComponentUpdate**
  available in class components

- For example, suppose **v1** and **v2** are variables
  whose values come from calls to **useState** or **useContext**
  and these are used in the calculation of JSX to be rendered

- To only calculate new JSX if one or both of them
  have changed since the last render,
  pass the JSX as the first argument to **useCallback**

- For example

```
return useCallback(
  <div>
    component JSX goes here
  </div>,
  [v1, v2]
);
```

- **useMemo** takes a function and
  an array of variables that affect the result

- Memoizes the function and returns its current result

- Only remembers result for last set of input values,
  not all past unique calculations

- Suppose **x** and **y** are variables whose values
  come from calls to **useState** or **useContext**
  and we need to compute a value based on these

- The following code reuses the previous result
  if the values of **x** and **y** have not changed

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

function MemoDemo() {
  const [x, setX] = useState(4);
  const [y, setY] = useState(3);

  const changeX = event => setX(event.target.value);
  const changeY = event => setY(event.target.value);

  const hypot = useMemo(() => {
    console.log('calculating hypotenuse');
    return Math.sqrt(x * x + y * y);
  }, [x, y]);

  return (
    <div>
      <input onChange={changeX} value={x} />
      <input onChange={changeY} value={y} />
      {hypot}
    </div>
  );
}

export default MemoDemo;
```

hypot is a value, not a function

... Memo Hook

- Note difference between **useCallback** and **useMemo**

- While both provide memoization,
  useCallback returns a value (which can be a function) and
  useMemo returns the result of calling a function

React.memo Function ...

- A function, not a hook

- Added in React 16.6

- Memoizes a function component so
  it is only re-rendered if at least one of its **props** has changed

- Does what class components do when they
  extend from **PureComponent** instead of **Component**

- For example, the following defines a **Percent** component
  that renders the percentage a count represents of a total

```
import React from 'react';                          percent.js

const Percent = React.memo(({count, total}) => {
  console.log('Percent rendering'); // to verify when this happens
  const percent = total === 0 ? 0 : (count / total) * 100;
  return <span>{percent.toFixed(2)}%</span>;
});


export default Percent;
```

... React.memo Function

- By default, performs a shallow compare of ALL props

- To customize comparison

  - pass a function as second argument

  - will be passed two objects containing the previous and next props

  - return **true** to skip render

```javascript
import React from 'react';

function skipRender(prevProps, nextProps) {
  return (
    prevProps.count === nextProps.count &&
    prevProps.total === nextProps.total
  );
}

const Percent = React.memo(({count, total, symbol}) => {
  console.log('Percent rendering'); // to verify when this happens
  const percent = total === 0 ? 0 : (count / total) * 100;
  return <span>{percent.toFixed(2)}{symbol}</span>;
}, skipRender);

export default Percent;
```

percent.js

re-renders if **count** or **total** change, but not if **symbol** changes

29

Reducer Hook ...

- **useReducer** supports implementing components whose state is updated by dispatching actions that are handled by a reducer function
  - patterned after Redux
  - takes reducer function and initial state
- Example
  - very simple todo app with a single component, **TodoList**
  - uses Sass for styling (nested rules)
  - calls **useReducer** to obtain **state** and **dispatch** function
  - calls **dispatch** in event handling functions
  - only re-renders when calls to **dispatch** result in a state change

Todos

| todo text | + |

- ☐ foo **X**
- ☑ ~~bar~~ **X**
- ☐ baz **X**

30

todo-list.scss

```scss
.todo-list {
  .delete-btn {
    background-color: transparent;
    border: none;
    color: red;
    font-weight: bold;
  }

  .done-true {
    color: gray;
    text-decoration: line-through;
  }

  form {
    margin-bottom: 10px;
  }

  .todo {
    margin-bottom: 0;
  }
}
```

31

```
import React, {useCallback, useReducer} from 'react';
import './todo-list.scss';                    todo-list.js

const initialState = {
  text: '',
  todos: []
  // objects in this have id, text, and done properties.
};

let lastId = 0;


// prevents form submit
const handleSubmit = e => e.preventDefault();
```

```
function reducer(state, action) {                           todo-list.js
  const {text, todos} = state;
  const {payload, type} = action;
  switch (type) {
    case 'add-todo': { // doesn't use payload
      const newTodos = todos.concat({id: ++lastId, text, done: false});
      return {...state, text: '', todos: newTodos};
    }
    case 'change-text':
      return {...state, text: payload};
    case 'delete-todo': {
      const id = payload;
      const newTodos = todos.filter(todo => todo.id !== id);
      return {...state, todos: newTodos};
    }
    case 'toggle-done': {
      const id = payload;
      const newTodos = todos.map(
        todo => (todo.id === id ? {...todo, done: !todo.done} : todo)
      );
      return {...state, todos: newTodos};
    }
    default:
      return state;
  }
}
```

33

```
export default function TodoList() {
  const [state, dispatch] = useReducer(reducer, initialState);

  const handleAdd = useCallback(
    () => dispatch({type: 'add-todo'}),
    []);

  const handleDelete = useCallback(
    id => dispatch({type: 'delete-todo', payload: id}),
    []
  );

  const handleText = useCallback(
    e => dispatch({type: 'change-text', payload: e.target.value}),
    []
  );

  const handleToggleDone = useCallback(
    id => dispatch({type: 'toggle-done', payload: id}),
    []
  );
```

todo-list.js

want same event handling function in each render, so second argument to `useCallback` is empty array

todo-list.js

```
  return (
    <div className="todo-list">
      <h2>Todos</h2>
      <form onSubmit={handleSubmit}>
        <label htmlFor="text">
          <input
            placeholder="todo text"
            onChange={handleText}
            value={state.text}
          />
        </label>
        <button onClick={handleAdd}>+</button>
      </form>
      {state.todos.map(todo => (
        <div className="todo" key={todo.id}>
          <input
            type="checkbox"
            checked={todo.done}
            onChange={() => handleToggleDone(todo.id)}
         />
          <span className={`done-${todo.done}`}>{todo.text}</span>
          <button className="delete-btn" onClick={() => handleDelete(todo.id)}>
            X
          </button>
        </div>
      ))}
    </div>
  );
}
```

Imperative Handle Hook

- **`useImperativeHandle`** modifies
the instance value parent components will see
if they obtain a ref to the current component

- One use is to add methods to the instance
that parent components can call

- Example

  - suppose current component contains multiple inputs

  - it could use this hook to add a method to its instance value
that parent components can call to move focus to a specific input

Layout Effect Hook

- **useLayoutEffect** is used to query and modify the DOM

- Similar to **useEffect**,
  but differs in that the function passed to it
  is invoked <u>after every DOM mutation in the component</u>

- DOM modifications are applied synchronously

- One use is to implement **animations**

Ref Hook ...

- **useRef** provides an alternative to using class component instance variables in function components

- Refs persist across renders

- Differ from capturing data with **useState** in that changes to their values do not trigger the component to re-render

- **useRef** optionally takes an initial value and returns an object whose **current** property holds current value

- **Common use** is to capture references to DOM nodes

  - can't use **React.createRef()** in function components

  - use **useRef()** instead
    ```
    const inputRef = useRef();
    return (
      <input ref={inputRef} onChange={onChange} value={value} />
    );
    // use inputRef.current somewhere
    ```

38

... Ref Hook ...

- Example: in Todo app above we can automatically move focus to text input

- Steps

  1. import **useEffect** and **useRef** hooks

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

  2. create ref inside component function

     ```
     const inputRef = useRef();
     ```

  3. add an effect to move focus

     ```
     useEffect(() => inputRef.current.focus(), []);
     ```

     executes only in first render

  4. set ref using **input** element **ref** prop

     ```
     <input
       placeholder="todo text"
       onChange={handleText}
       ref={inputRef}
       value={state.text}
     />;
     ```

39

... Ref Hook

- Ref values are not required to be DOM nodes

- Example: can log number of todos that have been deleted every time one is deleted

- Steps

    1. create ref inside component function to hold count

    2. increment ref value every time a todo is deleted

    3. log current ref value

```javascript
const deleteCountRef = useRef(0); // initial value is zero

// Modified version of handleDelete function above.
const handleDelete = useCallback(
  id => {
    dispatch({type: 'delete-todo', payload: id});
    deleteCountRef.current++;
    console.log('You have deleted', deleteCountRef.current, 'todos.');
  },
  [] // want same function on every render
);
```

Custom Hooks ...

- Functions whose name begins with "`use`"
  and call one more hook functions

- Typically return an array or object
  that contains state data and
  functions that can be called to modify the state

  - like provided `useState` hook

- Useful for extracting hook functionality from function components
  so it can be reused in multiple function components

… Custom Hooks …

- Example: Dan Abramov demonstrated a custom hook
that watches the browser window width

```
function useWindowWidth() {
  const [width, setWidth] = useState(window.innerWidth);
  useEffect(() => {
    // setup steps
    const handleResize = () => setWidth(window.innerWidth);
    window.addEventListener('resize', handleResize);
    return () => {
      // cleanup steps when using component is unmounted
      window.removeEventListener('resize', handleResize);
    };
  }, []); // only runs on first render
  return width;
}
```

> maintains this state separately for each component that calls this function

- To use in a function component

```
const width = useWindowWidth();
```

- Every time the window width changes
using components will be given the width and will re-render

42

... Custom Hooks

- Example: Dan Abramov demonstrated
  simplifies associating a state value with a form input

```
function useFormInput(initialValue) {
  const [value, setValue] = useState(initialValue);
  const onChange = e => setValue(e.target.value);
  // Returning these values in an object instead of an array
  // allows it to be spread into the props of an HTML input.
  return {onChange, value};
}
```

- To use this in a function component that
  renders an `input` element for entering a name

```
const nameProps = useFormInput('');
// current value is in nameProps.value
return (
  <input {...nameProps} />
);
```

- Removes need for custom event handling!

43

Use Debug Value Hook ...

- Provided hook **useDebugValue** displays a primitive value
  after a custom hook name in React DevTools
  when a component that uses the hook is selected

- Can pass a different value in each hook invocation

- Only displays last value passed

- Useful for debugging

... Use Debug Hook

- Example

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

function useFormInput(initialValue) {
  const [value, setValue] = useState(initialValue);
  useDebugValue('value = ' + value);
  const onChange = e => setValue(e.target.value);
  return {onChange, value};
}
```

same as previous code except for use of **useDebugValue**

from **useFormInput** hook

from **useState** hook

Third Party Hooks

- The React community has created and shared additional hooks

- Many are listed at https://nikgraf.github.io/react-hooks

- Mine are in npm

  - **top-state-hook** - for sharing state between function components

  - **context-easy** - for managing all app state in a single context

Wrap Up

- Hooks are a great addition to React!

- They make implementing components much easier

- For many developers they spell the end of implementing React components with classes

- Over time we will see much less use of higher-order components and render props

- Make it simpler!