Basic React Hooks

Pau Fernández Jose L. Muñoz-Tapia

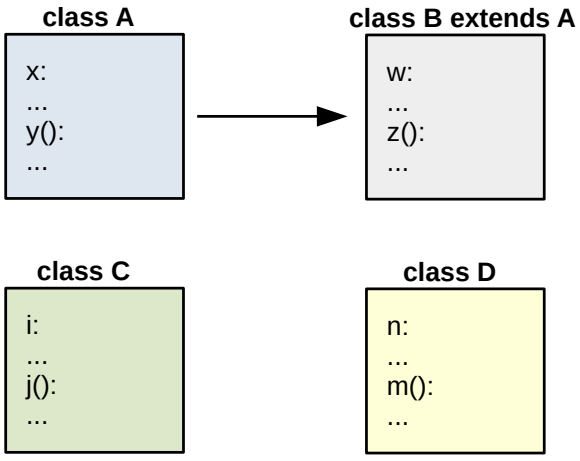
Universitat Politècnica de Catalunya (UPC)

Hooks

Outline

Reusing Object Methods

- With object oriented programming, many times it is hard to reuse methods from different inheritance hierarchies.
- E.g. to reuse methods of B and D we need **multiple inheritance**.
- React uses composition to try to avoid these problems.
- But still, it's hard to **reuse logic** between components.



Problems of Reusing Code in React

- Old solutions for reusing logic in React are:
  1. Higher Order Components (HOC)
  2. Render props refers to a technique for sharing code between React components using a prop whose value is a function.
- The problem is that these solutions impact the **component hierarchy**:
  - You might end with a ”wrapper hell” of components surrounded by layers of providers, consumers, higher-order components, render props, and other abstractions.
- Hooks will allow us to reuse stateful logic without changing your component hierarchy, **using a hook is an internal decision**.

Complexity

- Elaborated components become hard to understand:
  - Classes and its related stuff like binding and so on are more complex than functions.
  - We've often had to maintain components that started out simple but grew into an unmanageable mess of stateful logic and side effects.
- Life-cycle methods often contain a mix of unrelated logic:
  - Components might perform some data fetching in `componentDidMount` and `componentDidUpdate`.
  - The same `componentDidMount` method might also contain some unrelated logic that sets up event listeners, with cleanup performed in `componentWillUnmount`.

What are Hooks? How They Work?

- Hooks are just **functions**:
  - They let you use state and other React features without writing class components.
  - That is, with hooks we can **use only functional components**.
- Essentially, hooks work as follows:
  - You define your hooks at the top of your functional component.
  - React stores these hooks in an array.
  - React executes each hook in order when your component is mounted (or in general when defined for each particular hook).
- Hooks are a new addition in React 16.8[1].

[1] For compatibility, there are no plans to remove classes from React

Builtin Hooks

React provides several builtin Hooks[2].

- Basic Hooks
  - **useState**
  - **useEffect**
  - useContext
- Additional Hooks
  - useReducer
  - useCallback
  - useMemo
  - useRef
  - useImperativeHandle
  - useLayoutEffect
  - useDebugValue

[2] https://reactjs.org/docs/hooks-reference.html

Outline

useState

```jsx
export default class App extends React.Component {
 // state variable "count"
 constructor(props) {
   super(props);
   this.state = { count: 0 }; }

 render() {
  return ( <div>
     <p>You clicked {this.state.count} times</p>
     <button onClick={() =>
      this.setState(prevState => {count: prevState.count+1})}>
      Add
     </button>
     <button onClick={() => this.setState(0)}>Reset</button>
   </div> );
 }
}
```

```jsx
export default function App() {
  // state variable "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() =>
        setCount(prevCount => prevCount + 1)}>
        Add
      </button>
      <button onClick={() => setCount(0)}>Reset</button>
    </div>
  );
}
```

- The hook returns an array with exactly two elements:
  - The first element let's you access the state.
  - The second element (which is a function) let's you set the state.
- The state can be a boolean, number, string and object (also arrays).

Updates with **useState**

- As in class components, if the new state depends on the previous state you can use a function to set the state.
- But unlike the **setState** method found in class components, **useState does not automatically merge update objects**.
- You can replicate this behavior by combining the function updater form with object spread syntax:

```
setState( prevState => ({...prevState, ...updatedValues}) )
```

Lazy initial state

- The initialState argument is the state used during the initial render but in subsequent renders, it is disregarded.
- If the initial state is the result of an expensive computation:

```
// bad idea
const [state, setState] = useState(someExpensiveComputation(props));
```

- The expensive computation will be executed in each render.
- To fix this, you may provide a function to the hook instead.
- This function will be executed only on the initial render:

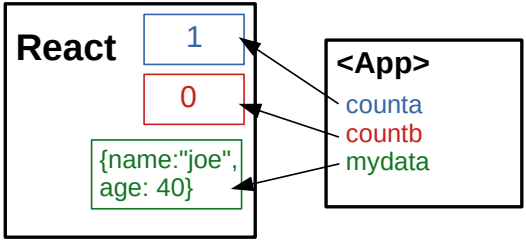
```
const [state, setState] = useState( () => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

Several Hooks

You can use several hooks on your functional components:

```
1  import React, { useState } from "react";
2
3  function App() {
4    const [counta, setCounta] = useState(0);
5    const [countb, setCountb] = useState(7);
6    const [mydata, setMyData] = useState({name:"joe", age: 40});
7    // ...
8  }
```

- Note that each state will be independent of the other states.
- Updating one state will therefore have no impact on the other states.
- React hooks use the order in which hooks are defined to link their state.



Rules of Hooks

You need to follow two rules when using hooks[3]:

1. Only call hooks at the top level:
   - Don't call Hooks inside loops, conditions, or nested functions.
   - Instead, always use Hooks at the top level of your React function.
   - By following this rule, you ensure that Hooks are called in the same order each time a component renders.
   - That's what allows React to correctly preserve the state with multiple Hooks[4].
2. Only call hooks from React functions:
   - You can call Hooks from React function components and from custom Hooks.
   - But don't call Hooks from regular JavaScript functions.

[3] The React community provides a linter plugin to enforce these rules automatically, `https://reactjs.org/docs/hooks-rules.html`

[4] React relies on the order in which Hooks are called to know which state corresponds to which hook.

Custom Hooks

You can create your custom hooks using other hooks:

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

function useMyCustomCounter(a,b){
  const [counta, setCounta] = useState(a);
  const [countb, setCountb] = useState(b);

  const increaseCounters = () => {
    setCounta(prevCount => prevCount+1);
    setCountb(prevCount => prevCount+2);
  }
  return [counta, countb, increaseCounters];
}

export default function App() {
  const [counta, countb, increaseCounters] = useMyCustomCounter(1,3);

  return (
    <div>
      <p>Counter a is: {counta} and Counter b is: {countb} </p>
      <button onClick={() => increaseCounters()}>  Increase  </button>
    </div>
  );
}
```

Outline

useEffect

- Using class components you can register a function on the `componentDidMount`, `componentWillUnmount` and `componentDidUpdate`.
- One very important feature of Hooks is allowing function components to have access to the lifecycle hooks.
- In particular, we have the hook `useEffect` for this purpose:
  1. The call accepts a function as argument.
  2. The function is executed when the component is first rendered and on every subsequent re-render/update.
  3. By default, effects run after every completed render but you can choose to fire them only when certain values have changed.
  4. React first updates the DOM, then calls any function passed to `useEffect` (without blocking the UI).
  5. To clean resources, the function passed to `useEffect` may return a clean-up function (which is executed on every effect).

Basic Usage of **useEffect**

We can **useEffect** to store our previous counter in local storage:

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

function App() {
  const [count, setCount] = useState( parseInt(JSON.parse(localStorage.getItem("count")), 10) || 0);
  useEffect( () => {
    localStorage.setItem("count", JSON.stringify(count));
    console.log("Effect called");
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(prevCount => prevCount + 1)}> Add </button>
      <button onClick={() => setCount(0)}> Reset </button>
    </div>
  );
}

export default App;
```

Better way for setting the initial state?

Triggering Effects

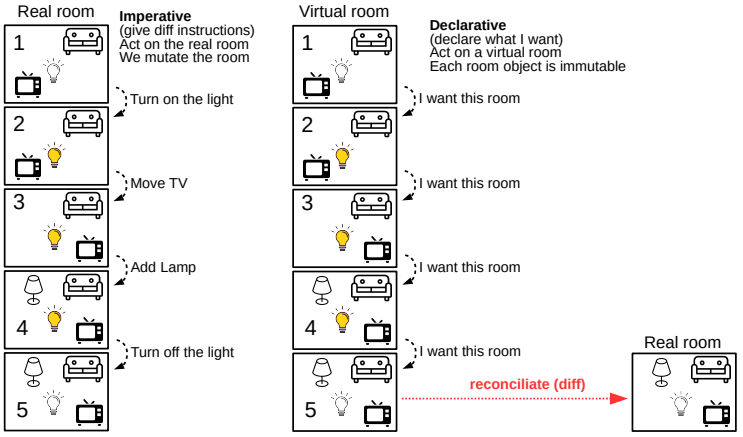
- Unlike `componentDidMount` or `componentDidUpdate`:
  - Effects scheduled with `useEffect` don't block the browser from updating the screen.
  - This makes your app feel more responsive[5].

- Important facts:
  1. The default behavior for effects is to **fire after every completed render**.
  2. Notice also that the **function passed to useEffect is going to be different on every render**.
     - Using different functions on every render is intentional.
     - This is what lets us read the count value from inside the effect without worrying about it getting stale (closure).

[5] In the uncommon cases where you need that effects happen synchronously there is a separate useLayoutEffect Hook with an API identical to useEffect.

Effects and Immutability

- Every time we re-render, we schedule a different effect, replacing the previous one.
- In a way, this **immutability** of effects makes them behave more like a part of the render result (each effect "belongs" to a particular render).



Every function inside the component render (including event handlers, effects, timeouts or API calls inside them) **captures the props and state of the render call that defined it**.

Why Effects Run on Each Update? i

- The reason is to avoid bugs.
- Consider a `<FriendStatus>` component that displays whether a friend is online or not.
- If we use a class component:
  - Our class reads friend.id from this.props.
  - Subscribes to the friend status after the component mounts.
  - Unsubscribes during unmounting.
- But what happens if the friend prop changes while the component is on the screen?

Why Effects Run on Each Update? ii

- Our component would continue displaying the online status of a different friend: this is a bug!
- In a class component:
  - We would need to add logic in `componentDidUpdate` to handle this.
  - But, forgetting to handle `componentDidUpdate` properly is a common source of bugs in React applications.
- `useEffect` does the function of the "update" check by default.

https://reactjs.org/docs/hooks-effect.html#explanation-why-effects-run-on-each-update

Issues with Performance of Effects

- Triggering updates after each rendering might create performance problems.
- In the following example we are using more effects than necessary:

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

export default function App() {
  const [counta, setCounta] = useState( parseInt(JSON.parse(localStorage.getItem("counta")), 10) || 0 );
  const [countb, setCountb] = useState(0);

  useEffect( () => {
      localStorage.setItem("counta", JSON.stringify(counta));
      console.log("Effect called");
  });
  return (
    <div>
      <p>You clicked a {counta} times</p>  <p>You clicked b {countb} times</p>
      <button onClick={() => setCounta(prevCount => prevCount + 1)}> Add a </button>
      <button onClick={() => setCountb(prevCount => prevCount + 1)}> Add b </button>
      <button onClick={() => setCounta(0)}> Reset a </button>
    </div>
  );
}
```

Skipping Effects

- With **useEffect** you can tell React to skip applying an effect if certain values haven't changed between re-renders.
- To do so, pass an array as an optional second argument to **useEffect**:

```
useEffect( () => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```

- We can pass an **empty array** [] as the second argument of **useEffect**:
  - Then, the effect is only fired once after component is mounted.
  - The props and state inside the effect will always have their initial values.

Cleaning up Effects

- Often, effects create resources that need to be cleaned up before the component leaves the screen, such as a subscription or timer ID.
- The function passed to `useEffect` can return a clean-up function:

```
1  import React, { useEffect, useState } from "react";
2  function MouseConsoleLogger() {
3    useEffect( () => {
4      const onMouseMove = event => { console.log(event); };
5      window.addEventListener("mousemove", onMouseMove);
6      return () => { window.removeEventListener("mousemove", onMouseMove); };
7    }, []);
8    return <div> </div>;
9  }
10
11 export default function App() {
12   const [toggle, setToggle] = useState(true);
13   return ( <div> Mouse Position Log in the Console
14           <button onClick={() => setToggle(!toggle)}> Toggle </button>
15           { toggle ? <MouseConsoleLogger/>: null }
16         </div> );
17 }
```

- React **cleans up the previous effect** before executing the new one.

Broken Example

```
1  import React, {useState, useEffect} from 'react';
2
3  export default function App() {
4
5      const [count, setCount] = useState(0);
6      const tick = () => {
7        console.log(`In tick, count: ${count}`);
8        setCount(count + 1);
9      }
10     useEffect( () => {
11       console.log("In useEffect");
12       const intervalID = setInterval(tick, 1000);
13       return () => {
14         console.log("Exitting effect");
15         clearInterval(intervalID);
16       }
17     }, []);
18   return (
19     <div>
20       Count: {count}
21     </div>
22     )
23 }
```

How to make the counter work?

++Prof Solution to Broken Example

- The problem: The closure of tick takes count = 0 all the time.
- Fix 1: make the effect depend on tick, however this is not clean.
- Fix 2:
  - Better to move tick into effect so the linter knows the variables that tick is using and we realize that we have to make it depend on count.
  - Linting rules will help us!
- Fix 3 (best):
  - Use the arrow function to update the state in `setCount`: `setCount(count => count+1)`
  - In this way the effect does not depend on any prop, state or derived variable from any of these.
  - We program the effect once and we do not create unnecessary functions that trigger unnecessary effects.
- To look: https://reactjs.org/docs/hooks-faq.html#is-it-safe-to-omit-functions-from-the-list-of-dependencies
- Note. We can say that for managing more complex cases we can use `useReducer`.

Outline

Trying to Use Hooks for Data Fetching

Next, we try to fetch data with React Hooks:

```
1  import React, { useState, useEffect } from 'react';
2  import axios from 'axios';
3
4  export default function App() {
5    const [data, setData] = useState({ hits: [] });
6    useEffect( async () => {
7      const result = await axios('https://hn.algolia.com/api/v1/search?query=ether');
8      setData(result.data);
9    });
10   return (
11     <ul>
12       {data.hits.map(item => (
13         <li key={item.objectID}>
14           <a href={item.url}>{item.title}</a>
15         </li>
16       ))}
17     </ul>
18   );
19 }
```

- ++Prof We will use the api of algolia.com to query hacker news:

```
$ curl
'https://hn.algolia.com/api/v1/search?query=ether'
```

- Based on https://www.robinwieruch.de/react-hooks-fetch-data.
- Note:
  - In the future, React Hooks are not be intended for data fetching in React.
  - Instead, a feature called </Suspense> will be in charge for it.
  - The following walk-through is nonetheless a great way to learn more about state and effect hooks in React.
- Effect runs when component mounts and when component updates:
  - Because we are setting the state after every data fetch, the component updates and the effect runs again.
  - It fetches the data again and again.
  - That's a bug and needs to be avoided.
  - We only want to fetch data when the component mounts.
  - That's why you can provide an empty array as second argument to the effect hook to avoid activating it on component updates but only for the mounting of the component.
  - Need to fix `useEffect(,[])`

Fixing Bugs

- There are two bugs in the previous code:
  1. We created a loop.
  2. The function passed as arg to **useEffect cannot be an async function**, it must return the cleaning function or nothing.

```
1  import React, { useState, useEffect } from 'react';
2  import axios from 'axios';
3
4  export default function App() {
5    const [data, setData] = useState({ hits: [] });
6    useEffect( () => {
7      const fetchData = async () => {
8        const result = await axios('https://hn.algolia.com/api/v1/search?query=ether');
9        setData(result.data);
10     };
11     fetchData();
12   }, []);
13   return (
14     <ul>
15       {data.hits.map(item => ( <li key={item.objectID}> <a href={item.url}>{item.title}</a> </li> ))}
16     </ul>
17   );
18 }
```

Triggering a Hook i

- Now, we will deal with error handling, loading indicators, triggering data fetching from a form and implementing a reusable ”data fetching hook”.
- To start, we want to use an input field to guide the fetching:

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

export default function App() {
  const [data, setData] = useState({ hits: [] });
  const [query, setQuery] = useState('ether');
  useEffect( () => {
    const fetchData = async () => {
      const result = await axios('https://hn.algolia.com/api/v1/search?query=${query}');
      setData(result.data); };
    fetchData(); }, []);

 return ( <>
   <input type="text" value={query} onChange={event => setQuery(event.target.value)} />
   <ul> {data.hits.map(item => ( <li key={item.objectID}> <a href={item.url}>{item.title}</a> </li> ))} </ul>
  </> );
}
```

Triggering a Hook ii

- One piece is missing:
  - When you try to type something into the input field, there is no other data fetching after the mounting triggered from the effect.
  - That's because you have provided the empty array as second argument to the effect.

```
export default function App() {
  const [data, setData] = useState({ hits: [] });
  const [query, setQuery] = useState('ether');

  useEffect(() => {
    const fetchData = async () => {
      const result = await axios(`https://hn.algolia.com/api/v1/search?query=${query}`);
      setData(result.data);
    };
    fetchData();
  }, [query]);
  // ...
}
```

- ++Prof The effect did not depend on variables, so it was only triggered when the component mounted.
- However, now the effect depends on the `query`.
- Once the `query` changes, the data request should fire again.

- Now, on every character you type into the input field, the effect is triggered and executes another data fetching request.

Triggering Manually

Let’s add a button button that triggers the request:

```
export default function App() {
  const [data, setData] = useState({ hits: [] });
  const [query, setQuery] = useState('ether');
  const [url, setUrl] = useState('https://hn.algolia.com/api/v1/search?query=ether');

  useEffect(() => {
    const fetchData = async () => {
      const result = await axios( `https://hn.algolia.com/api/v1/search?query=${query}`);
      setData(result.data);
    };
    fetchData();
  }, [url]);

  return ( <>
    <input type="text" value={query} onChange={event => setQuery(event.target.value)}/>
    <button type="button" onClick={() => setUrl(`https://hn.algolia.com/api/v1/search?query=${query}`)}> Search </button>
      <ul>
        {data.hits.map(item => ( <li key={item.objectID}> <a href={item.url}>{item.title}</a> </li> ))}
      </ul>
   </> );
}
```

Race Conditions

- Race conditions[6]:
  - We are fetching `{id: 10}`, switch to `{id: 20}`, but `{id: 20}` comes first.
  - The request that started earlier finished later and incorrectly overwrites my state (but I want the as latest state the one of the latest query).
- To fix this situation:
  - If the fetching process supports cancellation, we can use it in the cleanup.
  - Alternatively, we can use a boolean to track cancellations as follows:

```
1  function Article({ id }) {
2    const [article, setArticle] = useState(null);
3    useEffect(() => {
4      let didCancel = false;
5      async function fetchData() {
6        const article = await API.fetchArticle(id);
7        if (!didCancel) { setArticle(article); }
8      }
9      fetchData();
10     return () => { didCancel = true; }; }, [id]);
11   // ...
12 }
```

[6] Race conditions are typical in code that mixes async/await (which assumes something waits for the result) with top-down data flow (props or state can change while we're in the middle of an async function).

Adding a Loading Indicator

Let's introduce a loading indicator to the data fetching.

```
import React, { Fragment, useState, useEffect } from 'react';
import axios from 'axios';
export default function App() {
  const [data, setData] = useState({ hits: [] });
  const [query, setQuery] = useState('ether');
  const [url, setUrl] = useState('https://hn.algolia.com/api/v1/search?query=ether');
  const [isLoading, setIsLoading] = useState(false);
  useEffect(() => {
    const fetchData = async () => {
      setIsLoading(true);
      const result = await axios(url);
      setData(result.data);
      setIsLoading(false);
    };
    fetchData();
    }, [url]);

  return ( <>
    <input type="text" value={query} onChange={event => setQuery(event.target.value)}/>
    <button type="button" onClick={() => setUrl(`https://hn.algolia.com/api/v1/search?query=${query}`)}> Search </button>
    {isLoading ? ( <div>Loading ...</div> ) :
      ( <ul> {data.hits.map(item => ( <li key={item.objectID}> <a href={item.url}>{item.title}</a> </li> ))} </ul> )}
  </> );
}
```

- ++Prof. We just need another state that is managed by a state hook.
- Once the effect is called for data fetching, which happens when the component mounts or the URL state changes, the loading state is set to true.
- Once the request resolves, the loading state is set to false again.

Error Handling

```
export default function App() {
 const [data, setData] = useState({ hits: [] });
 const [query, setQuery] = useState('ether');
 const [url, setUrl] = useState('https://hn.algolia.com/api/v1/search?query=ether');
 const [isLoading, setIsLoading] = useState(false);
 const [isError, setIsError] = useState(false);
 useEffect(() => {
      const fetchData = async () => {
        setIsError(false);
        setIsLoading(true);
        try {
          const result = await axios(url);
          setData(result.data);
        } catch (error) { setIsError(true); }
        setIsLoading(false);
      };
      fetchData();
    }, [url]);

 return ( <>
    <input type="text" value={query} onChange={event => setQuery(event.target.value)} />
    <button type="button" onClick={() => setUrl(`https://hn.algolia.com/api/v1/search?query=${query}`)}> Search </button>
    {isError && <div>Something went wrong ...</div>}
    {isLoading ? ( <div>Loading ...</div> ) :
      ( <ul> {data.hits.map(item => ( <li key={item.objectID}> <a href={item.url}>{item.title}</a> </li> ))} </ul> )}
  </> );
}
```

Custom ”Data Fetching” Hook

```
const useHackerNewsApi = () => {
 const [data, setData] = useState({ hits: [] });
 const [url, setUrl] = useState('https://hn.algolia.com/api/v1/search?query=ether');
 const [isLoading, setIsLoading] = useState(false);
 const [isError, setIsError] = useState(false);
 useEffect(() => {
    const fetchData = async () => {
      setIsError(false);
      setIsLoading(true);
      try {
        const result = await axios(url);
        setData(result.data);
      } catch (error) { setIsError(true); }
      setIsLoading(false);
    };
    fetchData();
 }, [url]);
 return [{ data, isLoading, isError }, setUrl];
}
```

```
function App() {
 const [query, setQuery] = useState('ether');
 const [{data, isLoading, isError}, doFetch] = useHackerNewsApi();
 return (
  <>
   <form onSubmit={event => {
     event.preventDefault();
     doFetch(`https://hn.algolia.com/api/v1/search?query=${query}`);
   }}>
     <input
       type="text"
       value={query}
       onChange={event => setQuery(event.target.value)}
     />
    <button type="submit">Search</button>
   </form>
     // ...
  </> );
}
```

- We extract what belongs to the fetching process to a custom hook:
  - Note that the ”query” state is not extracted because it belongs to the input.
  - But we extract the loading indicator and error handling.