React hooks + components lifecycle + Http requests

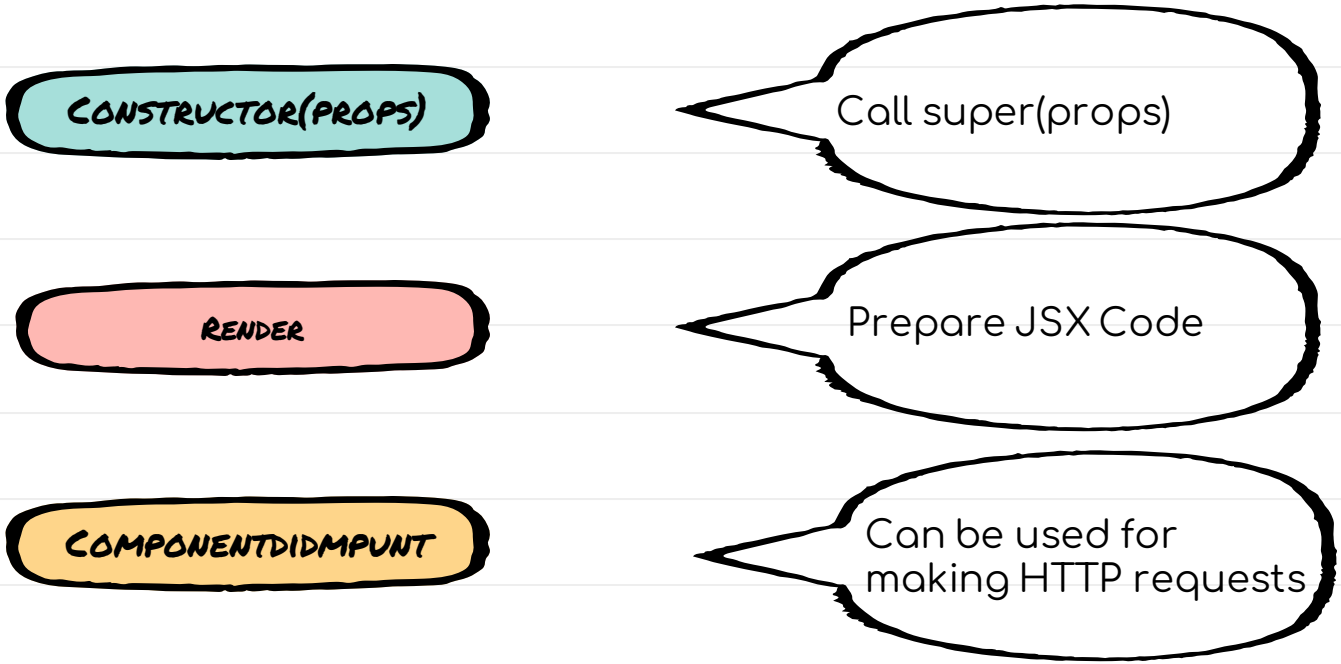
Teaching Faculty: Umur INAN

Prepared by Muhyieddin **AL-TARAWNEH**, Umur **INAN**

Component Lifecycle

- Each component has several “lifecycle methods” that you can override to run code at particular times in the process.
- Only available in Class-Based Components.

Component Lifecycle - Creation



Component Lifecycle - update

shouldComponentUpdate

May cancel update process

render

componentDidUpdate

Can be used for making HTTP requests

Component Lifecycle phases

- Render Phase
  - Pure and has no side effects.
  - Can be called multiple times by React.
  - Must be stable and return the same result.
    - Constructor
    - Render

Component Lifecycle phases

- Commit Phase
  - Run side effects.
  - Will be called once.
    - ComponentDidMount
    - ComponentDidUpdate
    - ComponentWillUnmount

Side effects

- Side effects are an action that impinges on the outside world.
  - Making asynchronous API calls for data .
  - Updating global variables from inside a function .
  - Working with timers like setInterval or setTimeout.
  - Logging messages to the console or other service.

componentDidMount

- When you use React, at a single point in time you can think of the render() function as creating a tree of React elements.
- On the next state or props update, that render() function will return a different tree of React elements.
- React then needs to figure out how to efficiently update the UI to match the most recent tree.

componentDidMount

- You may call setState() immediately in componentDidMount() (Most times, developers do that!).
- It will trigger an extra rendering, but it will happen before the browser updates the screen.
- This guarantees that even though the render() will be called twice in this case, the user won't see the intermediate state.
- Use this pattern with caution because it often causes performance issues.

componentDidUpdate

- componentDidUpdate() is invoked immediately after updating occurs.
  - This method is not called for the initial render.
- You may call setState() immediately in componentDidUpdate() but note that it must be wrapped in a condition, or you'll cause an infinite loop.

componentWillUnmount

- componentWillUnmount() is invoked immediately before a component is unmounted and destroyed.
- Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in componentDidMount().

React hooks

- Hooks are a new addition in React 16.8.
- They let you use state and other React features without writing a class.
- A Hook is a special function that lets you “hook into” React features..

When would I use a Hook?

- If you write a function component and realize you need to add some state to it, previously you had to convert it to a class.
- Now you can use a Hook inside the existing function component.
- To access Life Cycle Hooks.

Hook rules

- Only call Hooks at the top level. Don't call Hooks inside loops, conditions, or nested functions.
- Only call Hooks from React function components. Don't call Hooks from regular JavaScript functions.

useEffect

- Similar to
  - ComponentDidMount
  - ComponentDidUpdate
  - componentWillUnmount

useEffect

- ComponenentDidMount
  - If you want to run an effect, you can pass an empty array ([]) as a second argument.
  - This tells React that your effect doesn't depend on any values from props or state, so it never needs to re-run.

useEffect

- ComponenentDidUpdate / Conditional Update
  - 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

- componentWillUnmount
  - If your effect returns a function, React will run it when it is time to clean up.
  - Subscription, timer id, ...

- The clean-up function runs before the component is removed from the UI to prevent memory leaks.

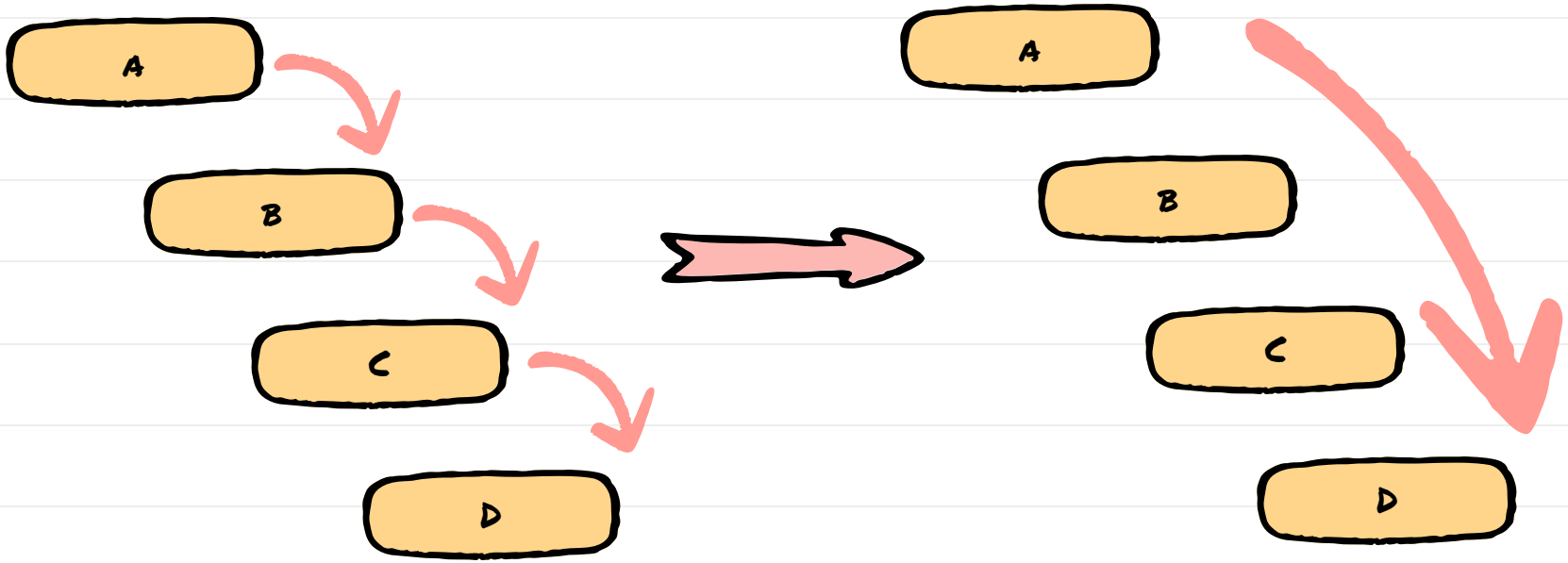
USECONTEXT

- In a typical React application, data is passed top-down (parent to child) via props, but such usage can be cumbersome.
- For example, passing locale preference to many components.
- React Context provides a way to pass data through the component tree without having to pass props down manually at every level.
- In other words, Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language.

BE CAREFUL

- Do not overuse! Every time the context value changes, all consumer components will rerender. That causes performance issues.
- Use for simple data that do not change often.
- If your state is frequently updated, React Context may not be as effective or efficient as a tool like Redux.

USECONTEXT



UseContext steps

- Create a context component.
    - When React renders a component that subscribes to this Context object it will read the current context value from the closest matching Provider above it in the tree.

```
const MyContext = React.createContext();
```

UseContext steps

- Wrap component/components with provider.

```
<MyContext.Provider value='someValue'>
    <App/>
</MyContext.Provider>
```

UseContext steps

- Retrieve the value from the one of the child components of <App>

```
const myContext= useContext(MyContext);
```

React Context vs Redux

- Redux is one of the most two popular state management tools in React.
- React Context and Redux both have the same goal, to manage global state so you don't need to pass down props to many components.
- They both create global variables like window.myVar.

React Context vs Redux

- The only difference is the way to access the global variable (state).
- If you use React Context properly and solve the re-rendering issues in child components, it will work just like Redux in big applications.
- Some developers don't like creating multiple folders and files in Redux. Other developers don't like writing producers and consumers in the component over and over again in React Context. So option is all up to you!

USEREF

- useRef returns a mutable ref object whose .current property is initialized to the passed argument (initialValue).
- The returned object will persist for the full lifetime of the component.
- A common use case is to access a child imperatively.
- useRef is like a “box” that can hold a mutable value in its .current property.

USEREF

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

Custom hooks

- Hooks are reusable functions.
- When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook.

Usefetch custom hook

```
const useFetch = (url, options) => {
  const [response, setResponse] = useState(null);
  useEffect(() => {
    const doFetch = async () => {
      const res = await fetch(url, options);
      const json = await res.json();
      setResponse(json);
    }
    doFetch();
  }, [url]);
  return response;
};
```

HTTP

- HTTP is the underlying protocol used by the World Wide Web and this protocol defines how messages are formatted, and what actions Web servers and browsers should take in response to various commands.

HTTP VERBS

- GET: Retrieves data from the server.
- HEAD: Same as GET, but response comes without the body.
- POST: Submits data to the server.
- PUT: Replace data on the server.
- PATCH: Partially update a certain data on the server.
- DELETE: Delete data from the server.
- OPTIONS: Handshaking and retrieves the capabilities of the server.

AXIOS

- Axios is a promise-based HTTP Client for node.js and the browser.
- Make XMLHttpRequests from the browser.
- By default, axios serializes JavaScript objects to JSON.
- Supports the Promise API Intercept request and response.
- Client-side support for protecting against XSRF.

Axios advantages

- Request and response interception
- Protection against XSRF
- Support for upload progress
- Response timeout
- The ability to cancel requests
- Support for older browsers
- Automatic JSON data transformation

Axios shorthand methods

- axios.request(config)
- axios.get(url[, config])
- axios.delete(url[, config])
- axios.post(url[, data[, config]])
- axios.put(url[, data[, config]])
- axios.patch(url[, data[, config]])

Axios -- POST

- Axios automatically serializes JavaScript objects to JSON when passed to the axios.post function as the second parameter. This eliminates the need to serialize POST bodies to JSON.
- Axios also sets the Content-Type header to application/json. This enables web frameworks to automatically parse the data.

Axios -- POST

```
axios.post('/users', {
    firstName: 'Fred',
    lastName: 'Flintstone'
  })
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

Axios.all

- Makes multiple requests in parallel by passing an array of arguments to the axios.all() method.

```
axios
.all(['url1', 'url2'])
.then(responseArray => {
    console.log(responseArray[0])
    console.log(responseArray[0])
}))
```

Axios config defaults

```
axios.defaults.baseURL = 'https://api.example.com';
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;
axios.defaults.headers.post['Content-Type'] = 'application/json';
```

Axios instance defaults

```
const instance = axios.create({
  baseURL: 'https://api.example.com'
});

instance.defaults.headers.common['Authorization'] = AUTH_TOKEN;
```

Main points

- REST gets access to information on the web through the ubiquitous URI. Everything on the web is available through a URI.
- Everything in creation is known through understanding and experience of the Unified Field of Consciousness.