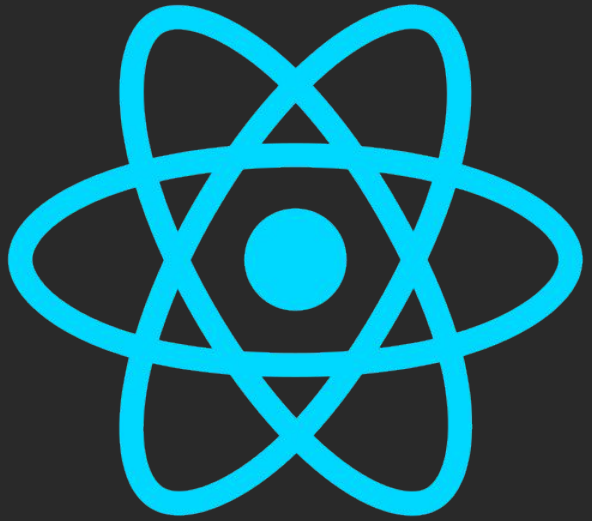
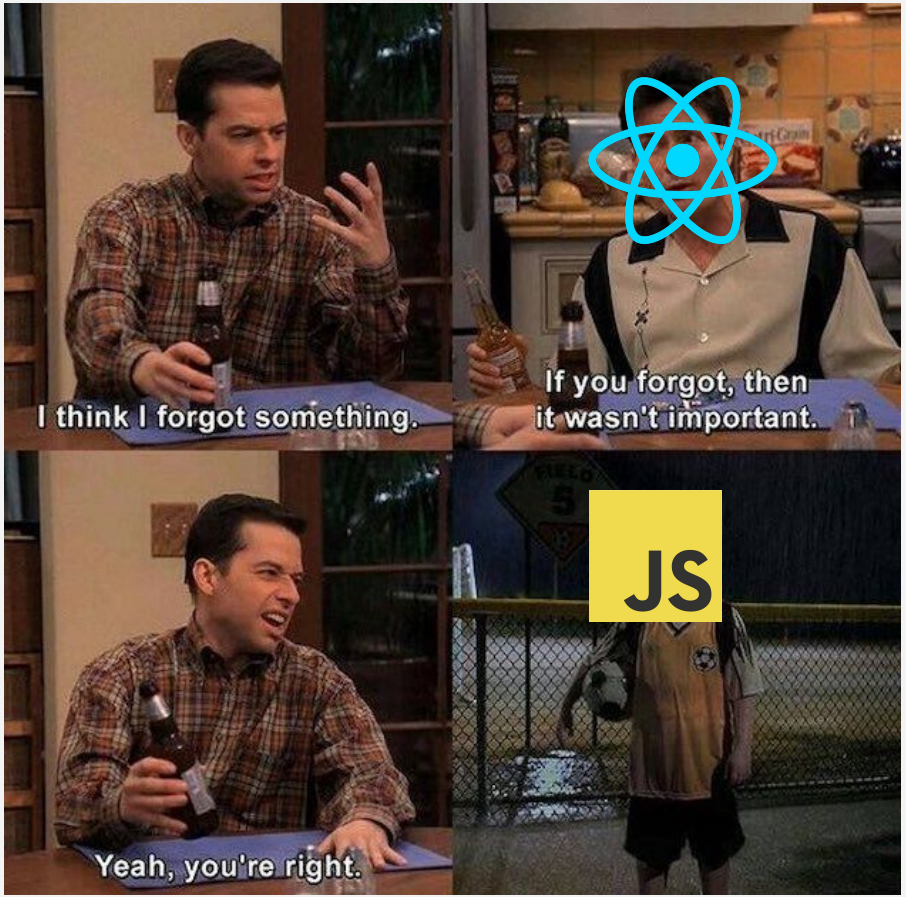


Asynchrony in React

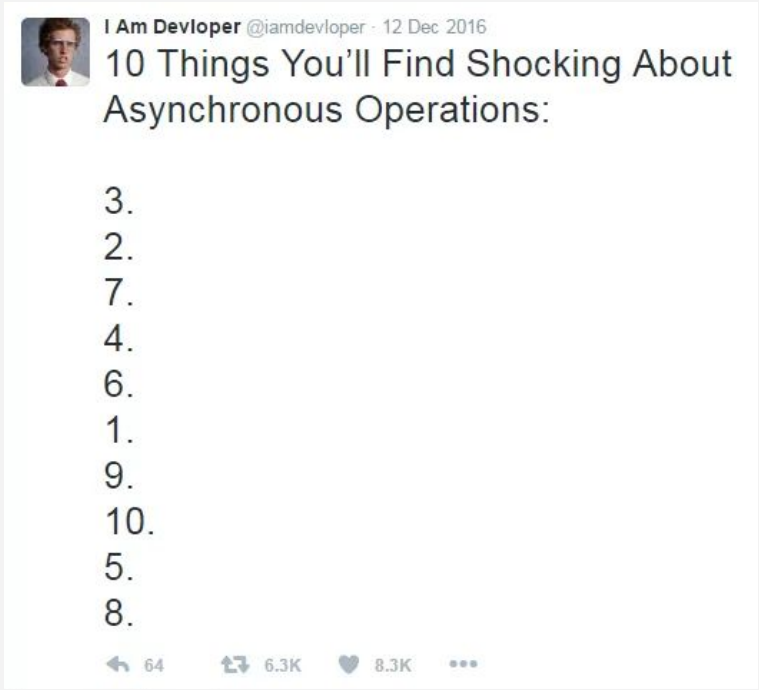
IMY 220 • Lecture 7

Incredibly Important Concept.



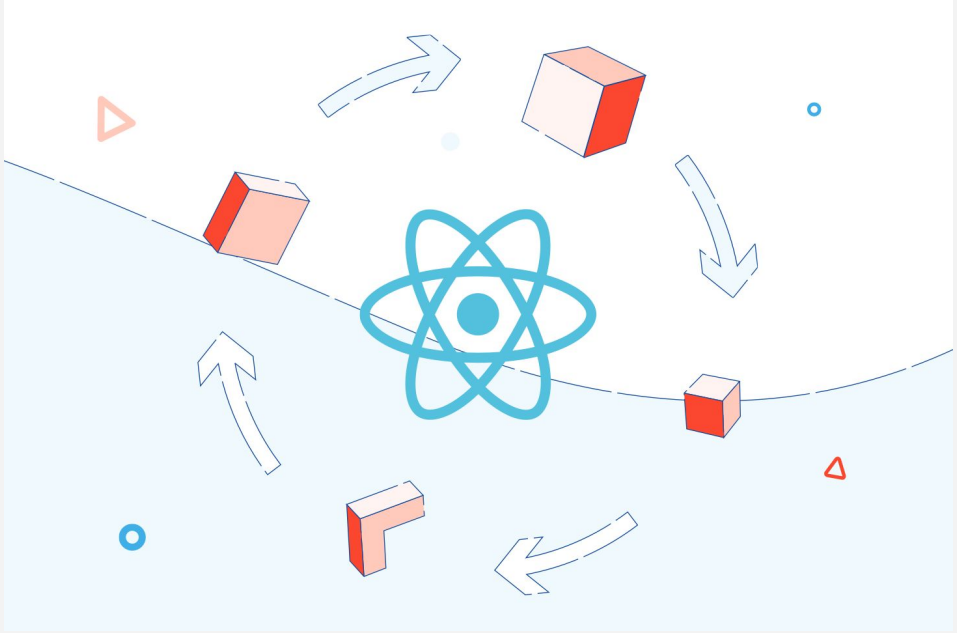
Async Behaviour in React

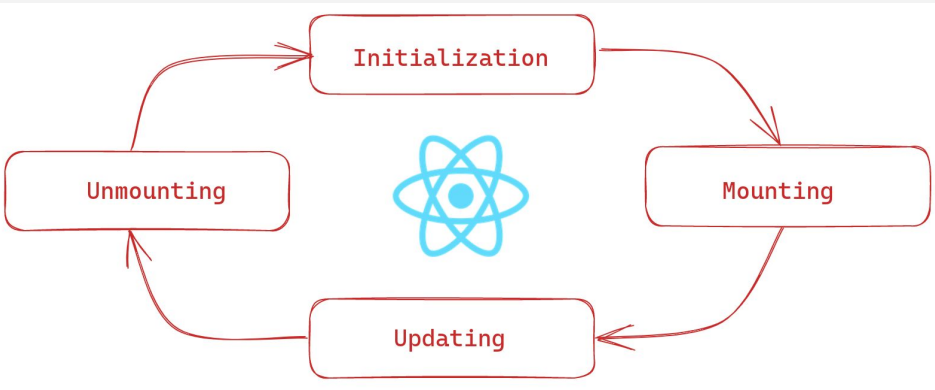
1. Component Lifecycle
2. Asynchronous Functions: `fetch()` API
3. Component State and Asynchrony
4. Duel-Wielding Asynchrony and Component State
5. Aside:
    a. State Management Libraries
    b. Server Components / SSR



Component Lifecycle

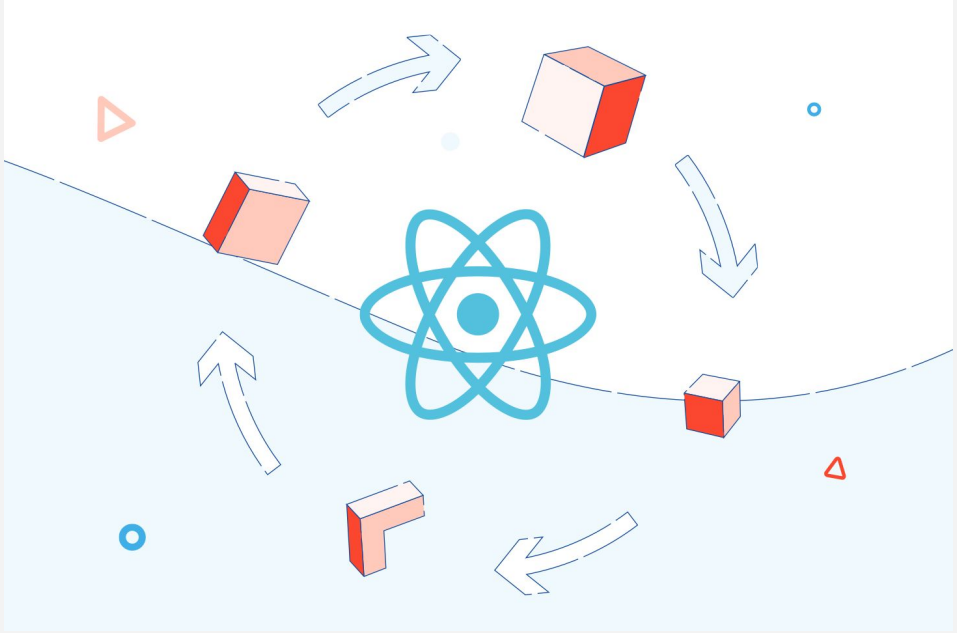
- Each React Component has a ‘lifecycle’
- We can execute code at each stage of the lifecycle by making use of special functions called **Lifecycle Methods**
  - Aka. Lifecycle **Hooks** in Functional Components

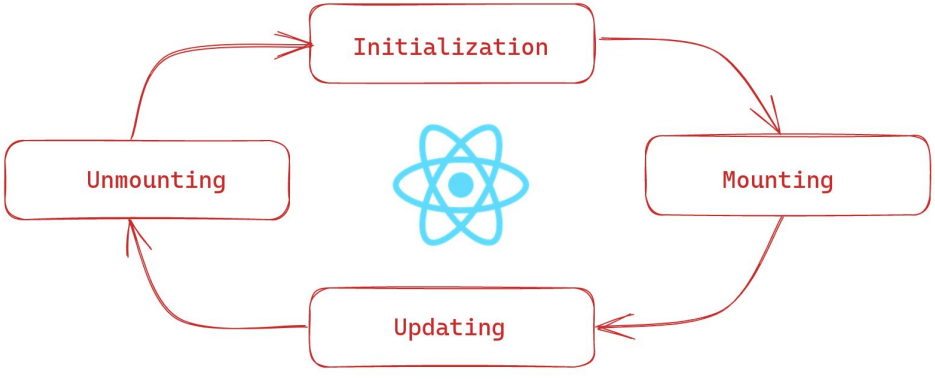




Component Lifecycle

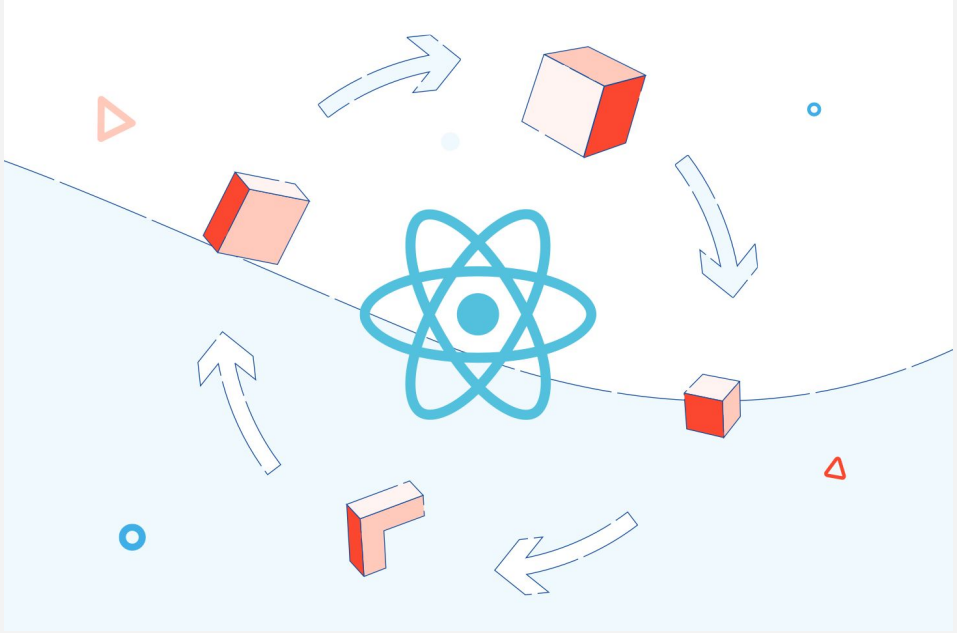
1. Initialisation (Creation)
2. Component Mount
    a. Added to the screen
3. Updating
4. Component Unmount
    a. Removed from the screen
5. Destruction (Cleanup)

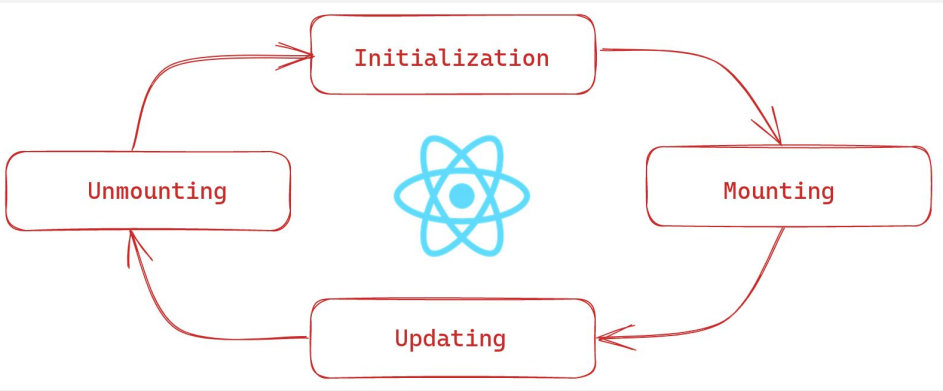




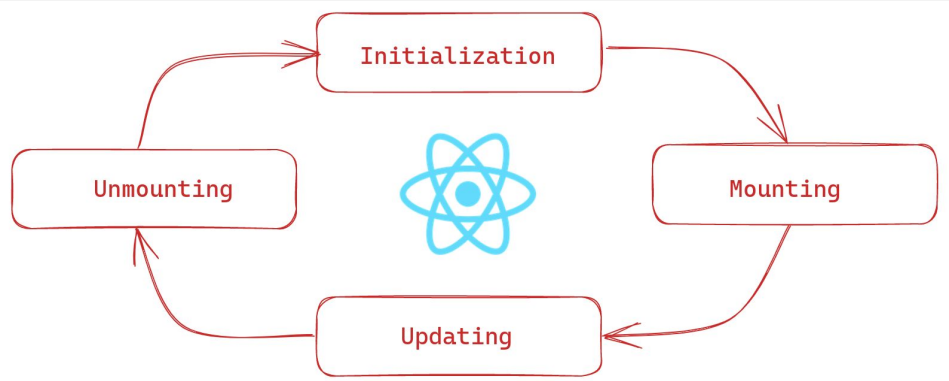
Component Lifecycle Methods

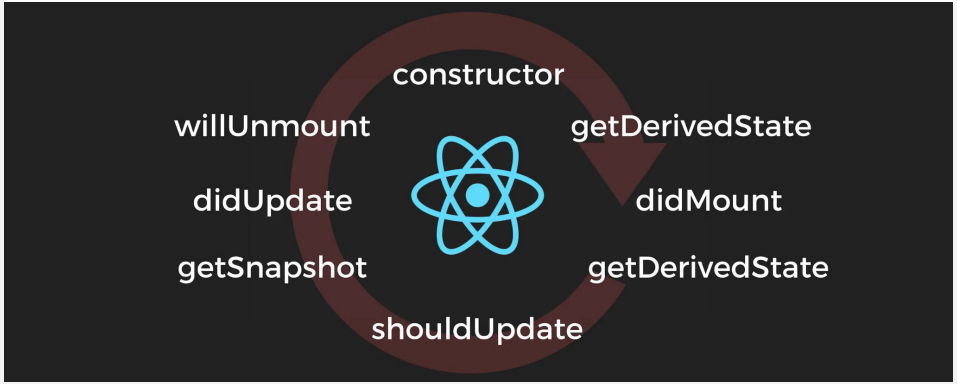
- React-defined component functions called at specific stages of the lifecycle.
  - e.g., Before / after a component is mounted and rendered.
- Different methods exist for Class and Functional Components.
  - Class = Lifecycle **Methods**
  - Functional = Lifecycle **Hooks**
  - If you want to study the differences in depth: https://react.dev/learn/lifecycle-of-reactive-effects





Class Component Lifecycle Methods



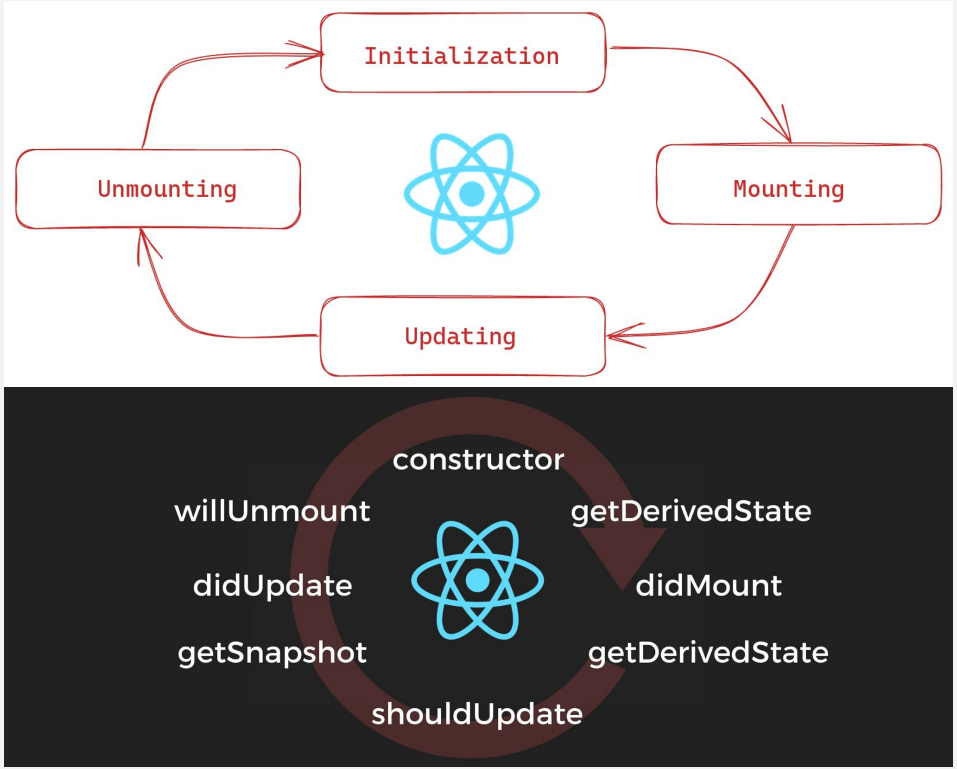


1. Initialisation (Creation)
   a. **`constructor()`**
2. Component Mount
   a. `getDerivedStateFromProps()`
   b. `render()`
   c. `componentDidMount()`
3. Updating
4. Component Unmount
5. Destruction (Cleanup)

There are more, but these are the main ones (common ones in **bold**).

React Docs:
https://legacy.reactjs.org/docs/react-component.html

Class Component Lifecycle Methods

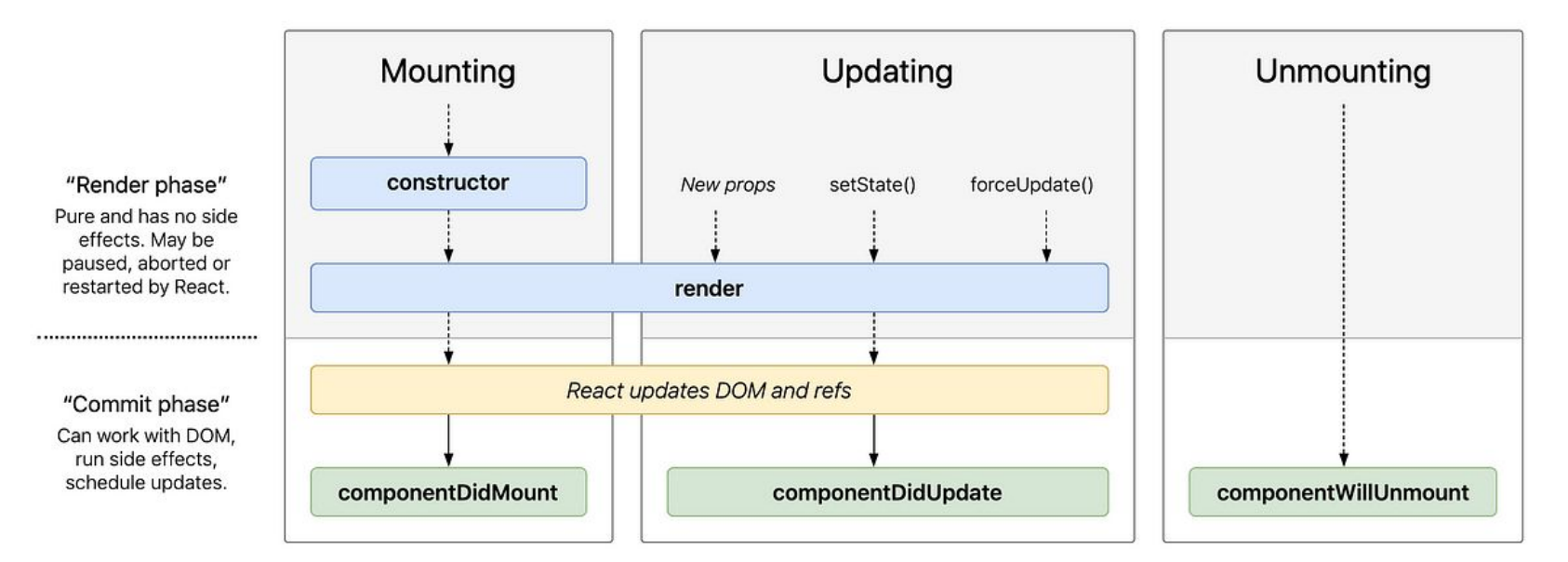


1. Initialisation
2. Component Mount
3. Updating
   a. `shouldComponentUpdate()`
   b. **`componentDidUpdate()`**
4. Component Unmount
   a. **`componentWillUnmount()`**
5. Destruction (Cleanup)

There are more, but these are the main ones (common ones in **bold**).

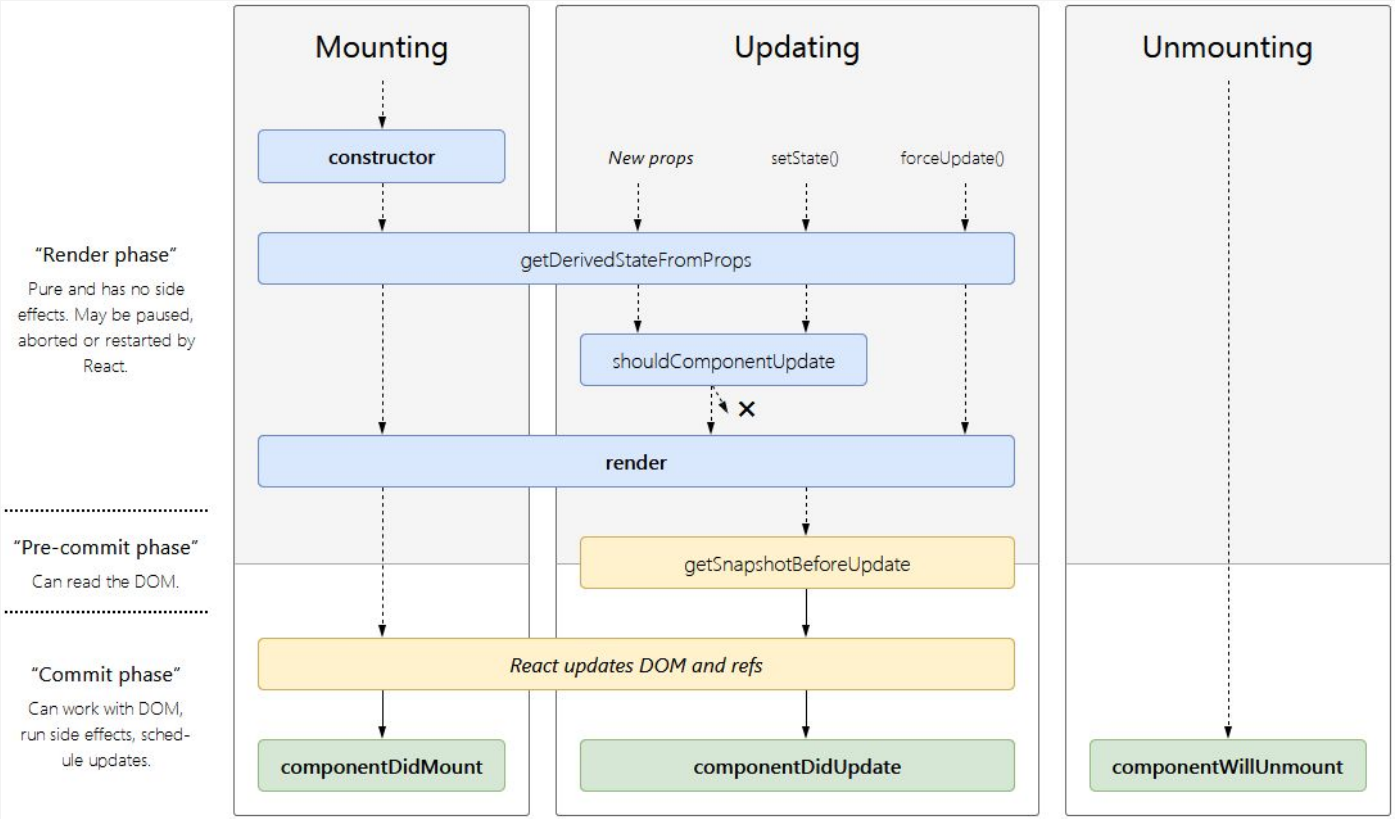
React Docs:
https://legacy.reactjs.org/docs/react-component.html

Class Component Lifecycle Methods



https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

Class Component Lifecycle Methods



https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

Whole list of Lifecycle Methods and Explanations in the React Docs:

https://legacy.reactjs.org/docs/react-component.html

https://react.dev/learn/render-and-commit

Read the Docs!

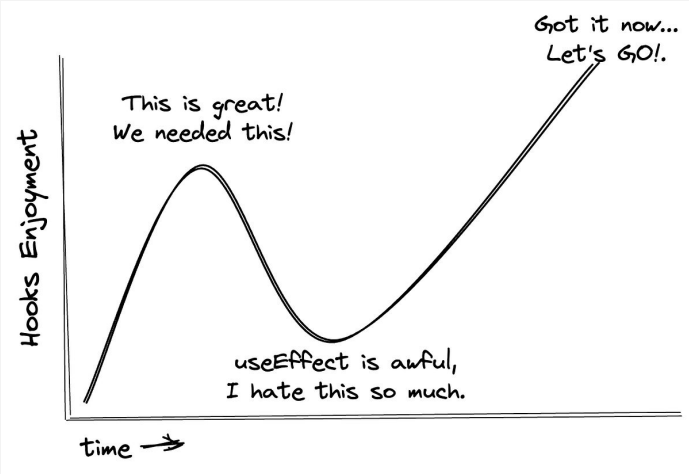
Class Component Lifecycle Methods

Code Demo

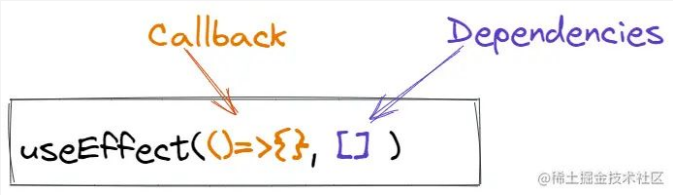
Functional Component Lifecycle Hooks

React Docs:
https://react.dev/learn/synchronizing-with-effects

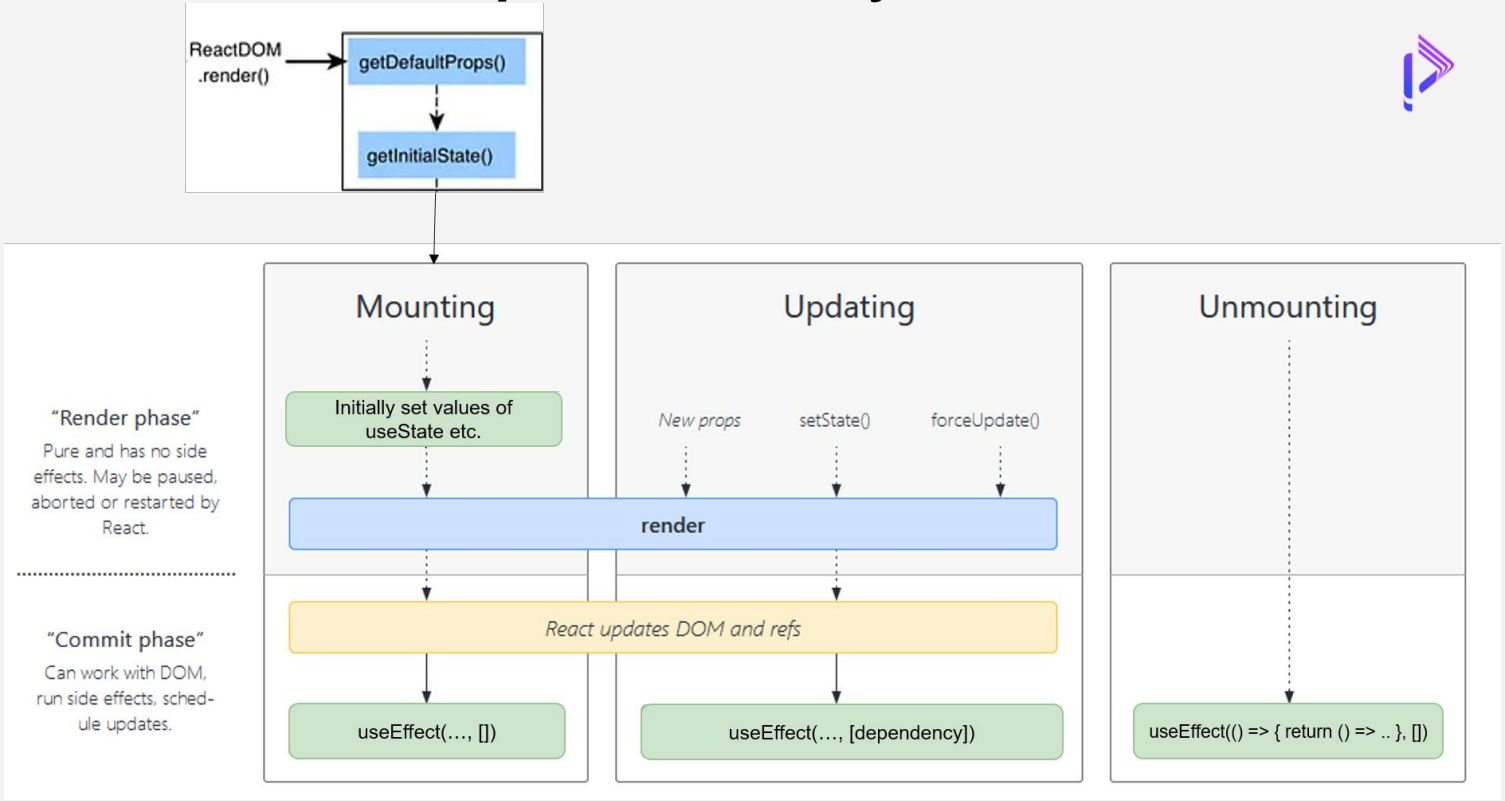
- **`useEffect()`**
  - `useEffect(callback, [dependency])`
  - Covers componentDidMount, componentDidUpdate and componentWillUnmount by using different variations of useEffect
- Used in combination with `useState()` and `setState()` to manage state updates
  - More Hooks, but these are most commonly used.



- Only supported in Functional Components, **not** Class Components.
  - React will throw an error if you try to use Hooks in Class Components
- Can also define your own custom Hooks
  - Typically use the `useX` convention



Functional Component Lifecycle Hooks



Whole list of Lifecycle Hooks and Explanations in the React Docs:

https://react.dev/learn/synchronizing-with-effects

Read the Docs!

https://github.com/ProgrammingHero1/react-functional-component-lifecycle

Functional Component Lifecycle Hooks

Code Demo

`fetch()` API

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

- Browser API used to make HTTP Requests
  - Introduced in 2015
  - Works with NodeJs as well (v.21+)
- Get data from the server without reloading the page
- **Native** to most modern browsers (standardized)
- Makes use of **Promises** & **Async** / **Await**
  - **Unlike** XMLHttpRequest, which uses callbacks
- Typically used for JSON, but can handle XML, HTML, etc.
- Seeker alternative to XMLHttpRequest API

`fetch()` API

Works using Promises

- Therefore we can use `.then()`, `.catch()` and `.finally()` just how we have already learned with ES6 Promises
- We can also use `async` and `await`

`fetch()` API

Structure: `fetch(url)`

- `url`: path to a resource (e.g., file, API) / `Request`
- Returns a `Promise` - resolves to `Response`
- GET request by **default**

- Also accepts HTTP **Headers** and **Body** by passing in a JS object to configure the request
- Get the response body as text via `.text()` or as JSON via `.json()`
- `.then()`= Executes when a Promise **resolves**
- `.catch()`= Executes when a Promise **rejects**
- `.finally()` = Executes regardless of what happens when a Promise **completes**

`fetch()` API: Making Requests

Using `.then()`, `.catch()`

```
const fetchData1 = () => {
  fetch('/api', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: "{ query: '{ hello }' }",
  })
    .then(res => res.json())
    .then(data => console.log(data))
    .catch(error => console.error(error));
};
```

Using `async` and `await` (with a `try/catch`)

```
const fetchData2 = async () => {
  try {
    const res = await fetch('/api', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: "{ query: '{ hello }' }",
    });

    const data = await res.json();
    console.log(data);

  } catch (error) {
    console.error(error);
  }
};
```

`fetch()` API: Handling Responses

Note:

- `.then()` **only** throws errors when:
  - **Network** errors e.g., timeouts
  - **CORS** errors
- `fetch()` will **resolve** for server status errors (e.g., 404 / 500)
  - Therefore `.catch()` will **not** execute due to server errors

To handle this, we can use the response returned from fetch to check status codes, etc.

- If we return a response object from fetch, e.g., `const res = await fetch(...);`
- `res.ok`
  - Returns `true` if response status code is between 200-299
  - `False` otherwise
- `res.status`
  - Returns the status code (e.g., `404`, `500`, `200`, etc.)

`fetch()` API: Handling Responses

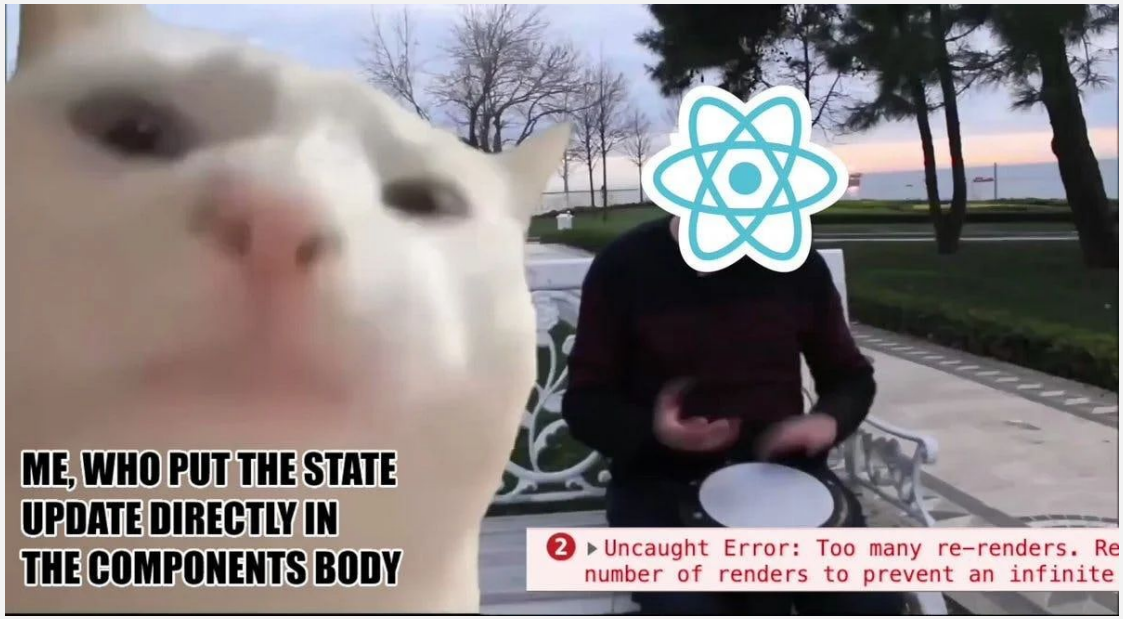
Example:

```
fetch('/api')
  .then(res => {
    if (!res.ok) {
      throw new Error('Network response was not ok');
    }

    if(res.status === 401) {
      throw new Error('Unauthorized');
    }

    return res.json();
  })
  .then(res => console.log(res.data))
  .catch(error => console.error(error));
```

```
try {
  const res = await fetch('/api');

  if (!res.ok) {
    throw new Error('Network response was not ok');
  }

  if(res.status === 401) {
    throw new Error('Unauthorized');
  }

  const data = await res.json();
  console.log(data);

} catch (error) {
  console.error(error);
}
```

Component State and Asynchrony

- Let’s put state updates and `fetch()` together **the wrong way**
- Code Demo



Component State

Components can have **state**

Several things can **trigger** state change / re-render.

Oftentimes want to update state with **API** data.

State is safely updated during the “Commit” phase - i.e., between completed renders.

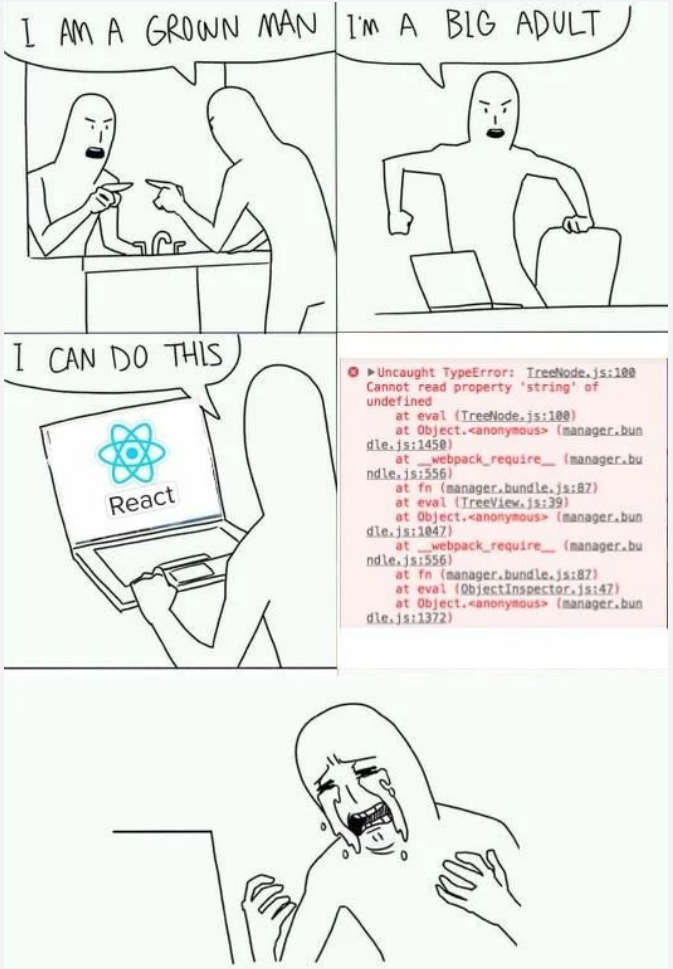
Component rendering itself should be **pure**

- If the state changes between the time when a component is created and before it mounts, or while a component is busy rendering, React will throw an error.

React Docs:
https://react.dev/learn/render-and-commit

“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.



Component State

React Docs:
https://react.dev/learn/render-and-commit

From their documentation:

Pitfall

Rendering must always be a pure calculation:

- **Same inputs, same output.** Given the same inputs, a component should always return the same JSX. (When someone orders a salad with tomatoes, they should not receive a salad with onions!)
- **It minds its own business.** It should not change any objects or variables that existed before rendering. (One order should not change anyone else's order.)

Otherwise, you can encounter confusing bugs and unpredictable behavior as your codebase grows in complexity. When developing in "Strict Mode", React calls each component's function twice, which can help surface mistakes caused by impure functions.

"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.

Component State and Asynchrony

- **You are likely to run into issues when dealing with React State and Asynchronous behaviour**.

Why?

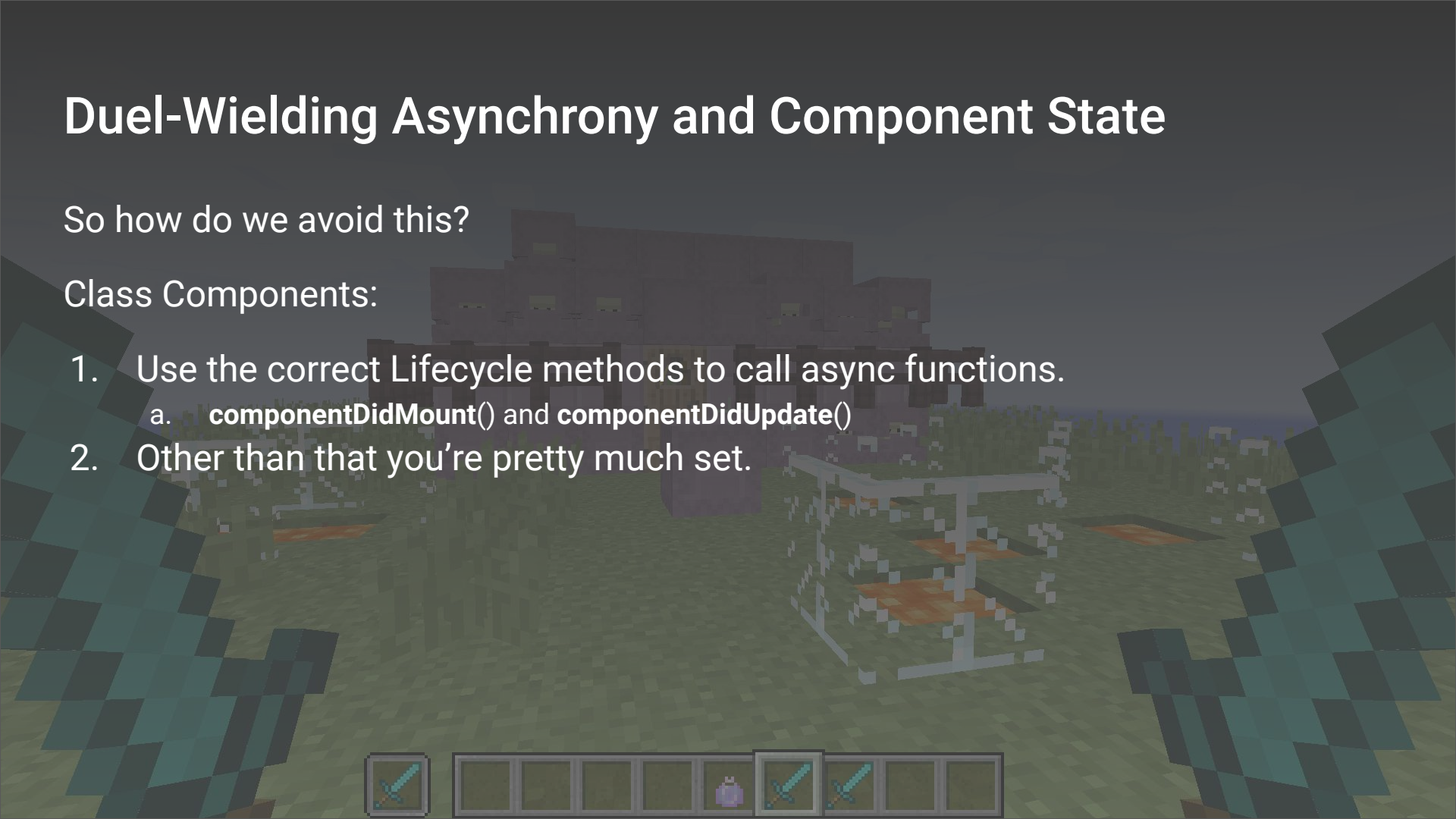
- If you accidentally ‘dirty’ (change) the component state during the ‘render’ phase, you cause unwanted side effects, which makes React throw an Error.
- **More often than not this comes up in Functional Components.**
  - Since it’s pretty easy to tell when Class lifecycle methods are being called.
  - `useEffect()` requires you understand a lot more about how state is updated.
  - Functional components are also beginning to disallow asynchronous behaviour in components using Hooks (in favour of Server-Side Rendering).

Duel-Wielding Asynchrony and Component State

So how do we avoid this?

Class Components:

1. Use the correct Lifecycle methods to call async functions.
    a. **componentDidMount**() and **componentDidUpdate**()
2. Other than that you're pretty much set.



Duel-Wielding Asynchrony and Component State

So how do we avoid this?

Functional Components:

1. Defining and using async functions immediately within useEffect() Hooks
    a. Trick React into letting you fetch async data and update state.
2. ‘Loading’ state management
    a. Prevent content from rendering until data is loaded.
    b. Triggering a ‘loaded’ state which then re-renders JSX with loaded data.

Resources for doing this properly - understanding state and Hooks:

https://www.youtube.com/watch?v=RAJD4KpX8LA

https://www.youtube.com/watch?v=Huq9LWPXr34

https://www.youtube.com/watch?v=-ylsOPp31L0

Only if you plan on using Hooks instead of Class Lifecycle Methods

Duel-Wielding Asynchrony and Component State

Code demo

A Warning About Console.log()

- We are pretty used to using console.log() to debug our code / check the state of our application at a point in time
- Be careful about using console.log() with async code
  - Console logs likely won't occur when you expect them to
  - Either use an `await` statement or place console.log()s in a `.then()` or `.catch()` statement.

```
const response = fetch('/api');
console.log(response);
```

```
const response = await fetch('/api');
console.log(response);
```

```
fetch('/api')
  .then(response => console.log(response));
```

A Warning About Console.log()

- Similarly to Async functions, be careful when using console.log() to check a component state.
- Why?
  - Component setState() may not be finished executing (due to it being “kind-of” async)
  - Why “kind of”?
    - Component .setState() is **synchronous**
    - BUT it **triggers** a bunch of asynchronous functionality (like updating the DOM, etc.)
    - Additionally multiple setState() updates are **batched**
    - This gives it the **appearance** of being asynchronous even when it isn’t
    - That means console.log() may execute before state updates fully.

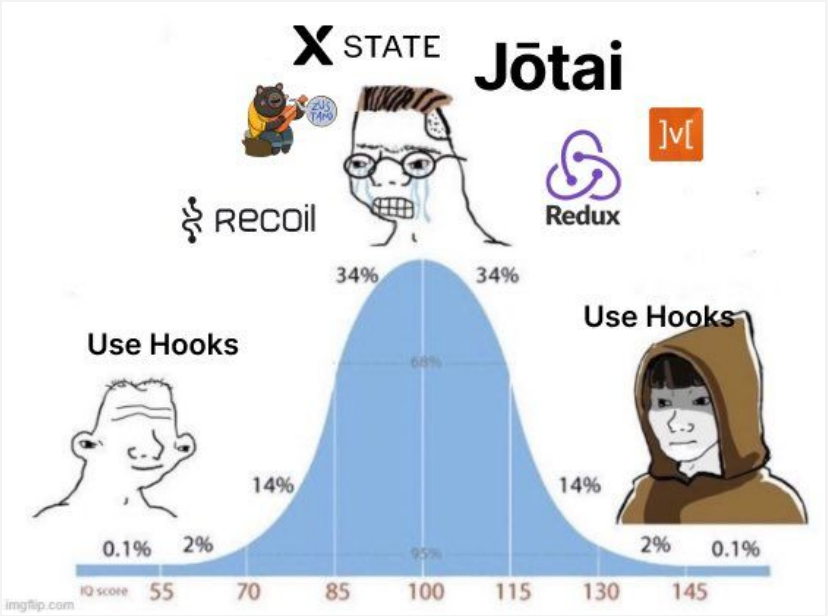
```
this.setState(count => ({ count: count + 1 }));
console.log(this.state.count);
```

Aside: State Management Libraries

- Many libraries that manage React state
- Used for complex state management
- Used instead of Lifecycle Methods / Hooks
- e.g., React Redux, Recoil, Zustand

For our project, you won't need to make use of any of these libraries. Hooks and regular state management will be fine.

Good to know about.



Aside: Server Components / SSR

Server-Side Rendering (SSR)

- Where components are instead rendered on the server, and then sent to the client as fully-rendered HTML
- **Server Components**
- Used in frameworks such as NextJs
  - Support coming to vanilla React soonish
- Support Asynchronous functionality
- Don’t Support State Management

Useful to know about.

References

Lifecycle:

https://legacy.reactjs.org/docs/react-component.html

https://react.dev/learn/render-and-commit

https://react.dev/learn/synchronizing-with-effects

https://react.dev/learn/lifecycle-of-reactive-effects

References

Fetch:

https://dev.to/dionarodrigues/fetch-api-is-new-old-version-of-ajax-1m14

https://dev.to/dionarodrigues/fetch-api-do-you-really-know-how-to-handle-errors-2gj0

https://www.geeksforgeeks.org/difference-between-ajax-and-fetch-api/

https://dev.to/dionarodrigues/fetch-api-is-new-old-version-of-ajax-1m14

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

References

Async:

https://www.youtube.com/watch?v=-ylsQPp31L0

https://www.youtube.com/watch?v=Huq9LWPXr34

https://www.youtube.com/watch?v=RAJD4KpX8LA

https://www.youtube.com/watch?v=zvM_FUVcB-0

Practise!