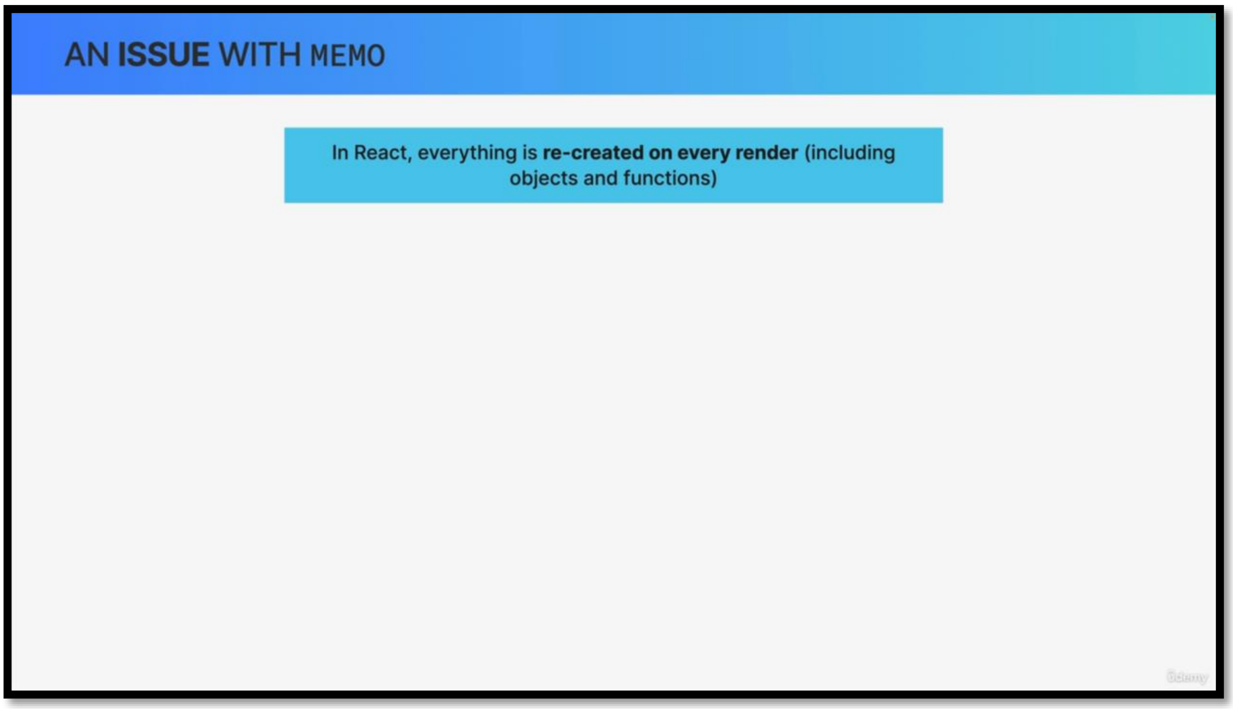


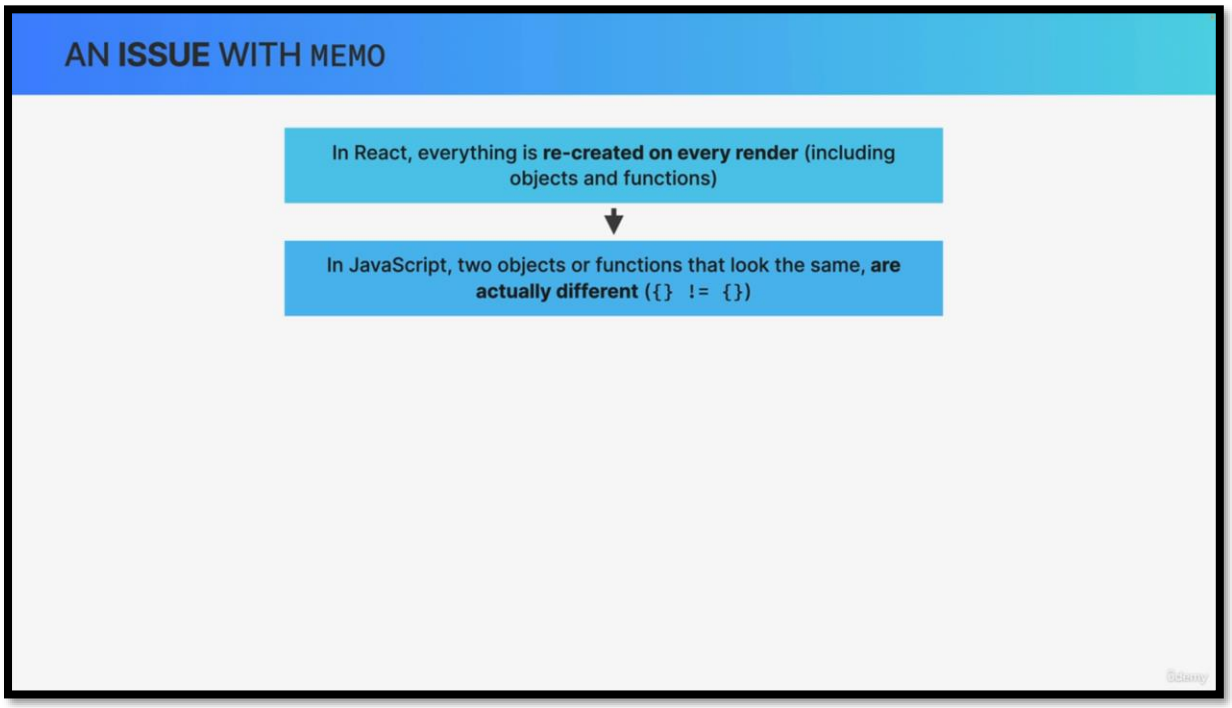
AN ISSUE WITH MEMO

In React, everything is **re-created on every render** (including objects and functions)

We know that in React whenever a component instance re-renders everything in there is recreated.

So, all values are always created again and that includes objects and functions that are defined within the component.

So, a new render gets new functions and new objects even if they are the exact same ones as before.

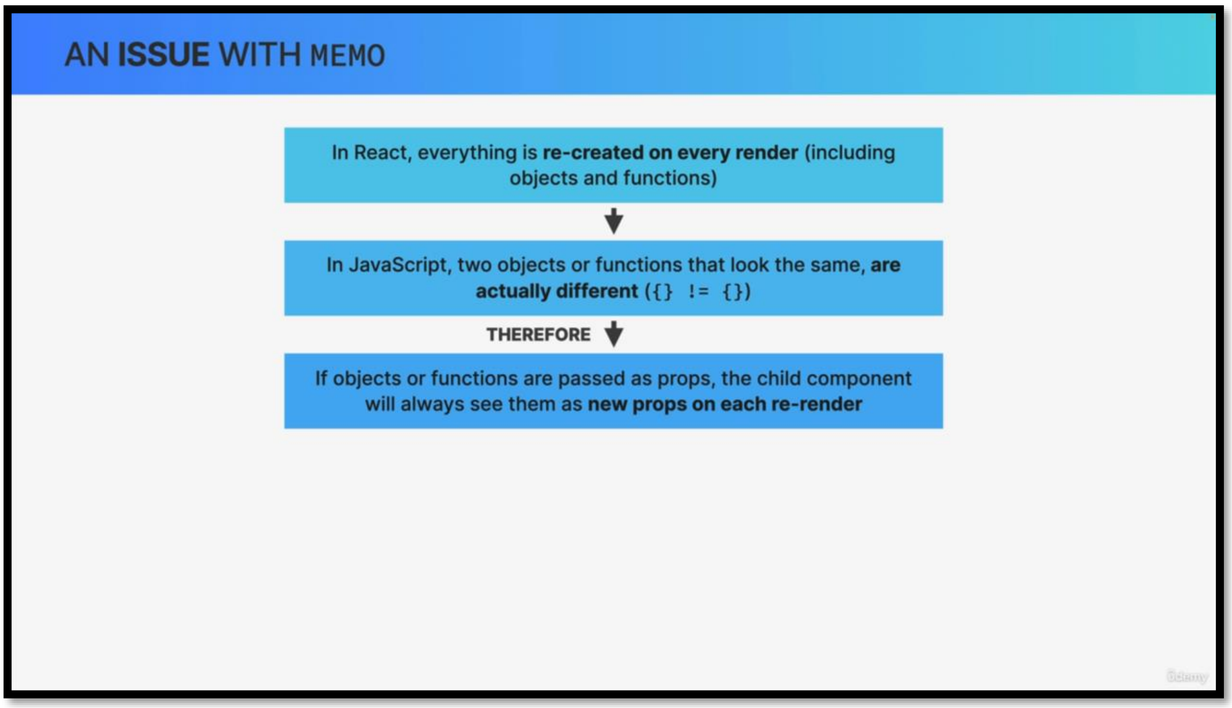


AN ISSUE WITH MEMO

In React, everything is **re-created on every render** (including objects and functions)

In JavaScript, two objects or functions that look the same, **are actually different** (`{}` `!=` `{}`)

In JavaScript two objects or functions that look the same, so that are exactly the same code are actually different unique objects. The classic example here is that an empty object is different from another empty object.



AN ISSUE WITH MEMO

In React, everything is **re-created on every render** (including objects and functions)

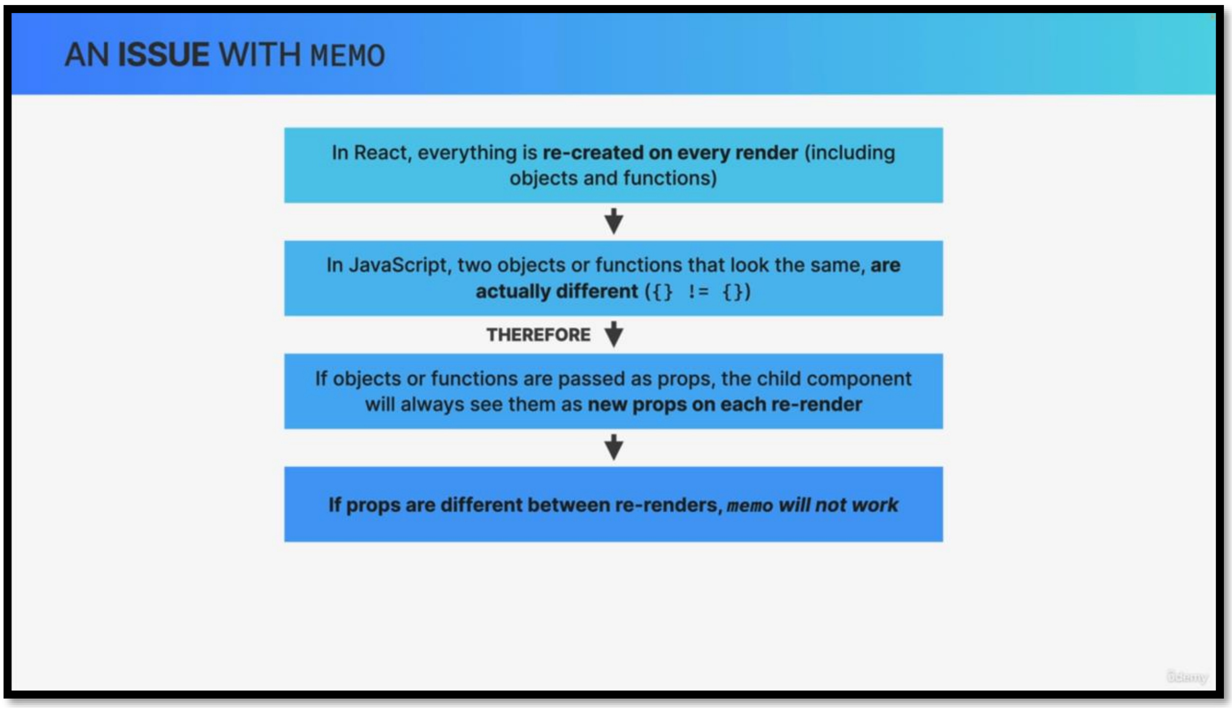
In JavaScript, two objects or functions that look the same, **are actually different** (`{} != {}`)

THEREFORE

If objects or functions are passed as props, the child component will always see them as **new props on each re-render**

Now, from these two pieces of information, we can understand that if we pass a function or an object to a child component as a prop, that child component will always see them as new props whenever there is a re-render.

If props are different between re-renders then memo will simply not work, so it will not do its job.


AN ISSUE WITH MEMO

In React, everything is **re-created on every render** (including objects and functions)

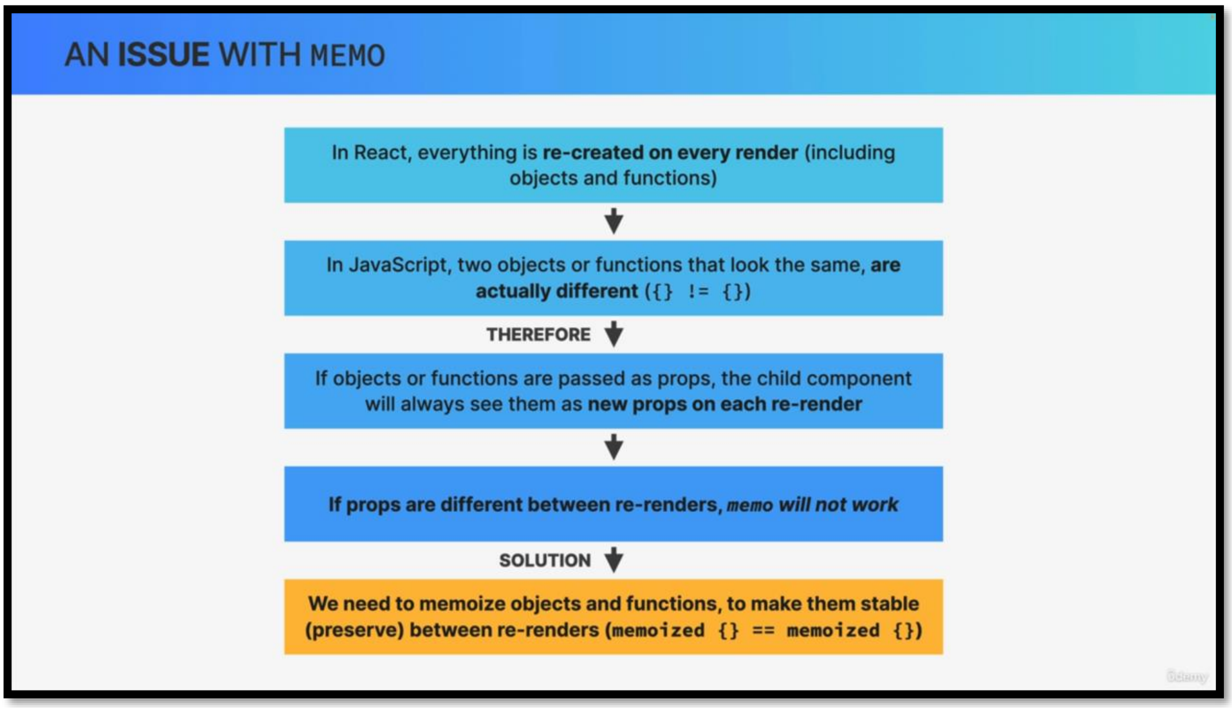
In JavaScript, two objects or functions that look the same, **are actually different** (`{} != {}`)

THEREFORE

If objects or functions are passed as props, the child component will always see them as **new props on each re-render**

If props are different between re-renders, *memo will not work*

So, in summary, if we memoize a component but then give it objects or functions as props, the component will always re-render anyway because it'll always see these props as new props, even when they actually look exactly the same.


AN ISSUE WITH MEMO

In React, everything is **re-created on every render** (including objects and functions)

In JavaScript, two objects or functions that look the same, **are actually different** (`{} != {}`)

THEREFORE

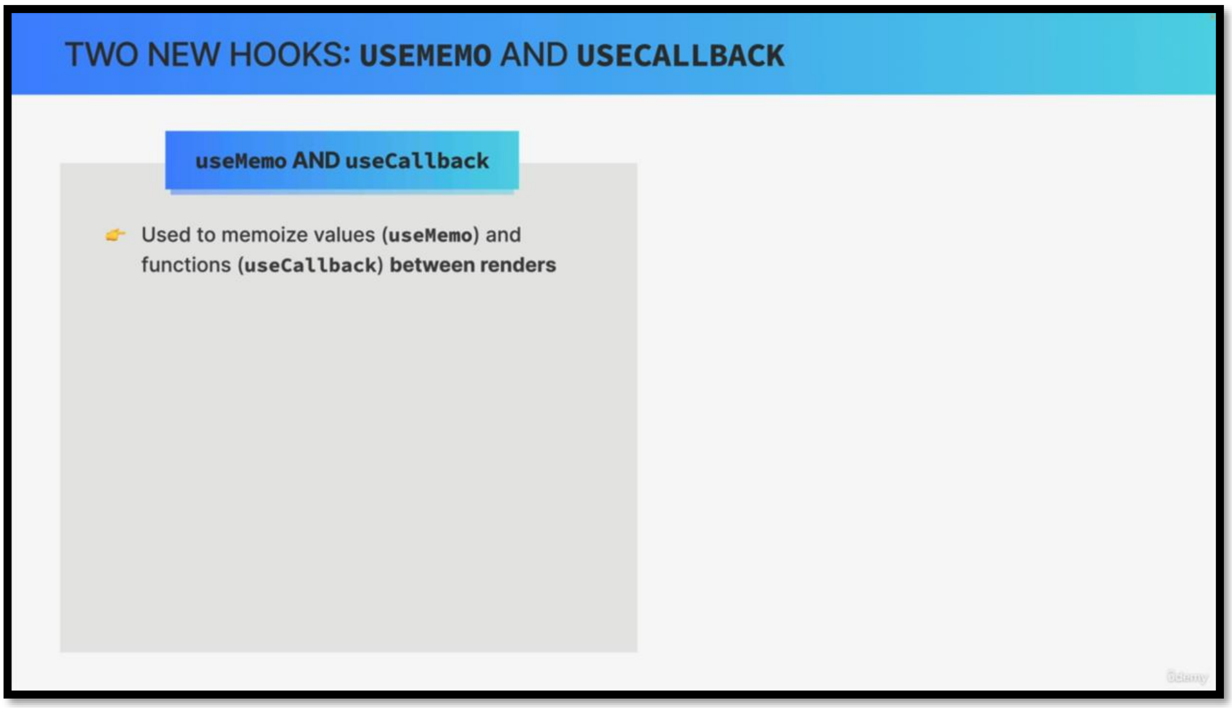
If objects or functions are passed as props, the child component will always see them as **new props on each re-render**

If props are different between re-renders, *memo will not work*

SOLUTION

We need to memoize objects and functions, to make them stable (preserve) between re-renders (`memoized {} == memoized {}`)

We can make objects and functions stable so we can actually preserve them between renders by memorizing them as well.



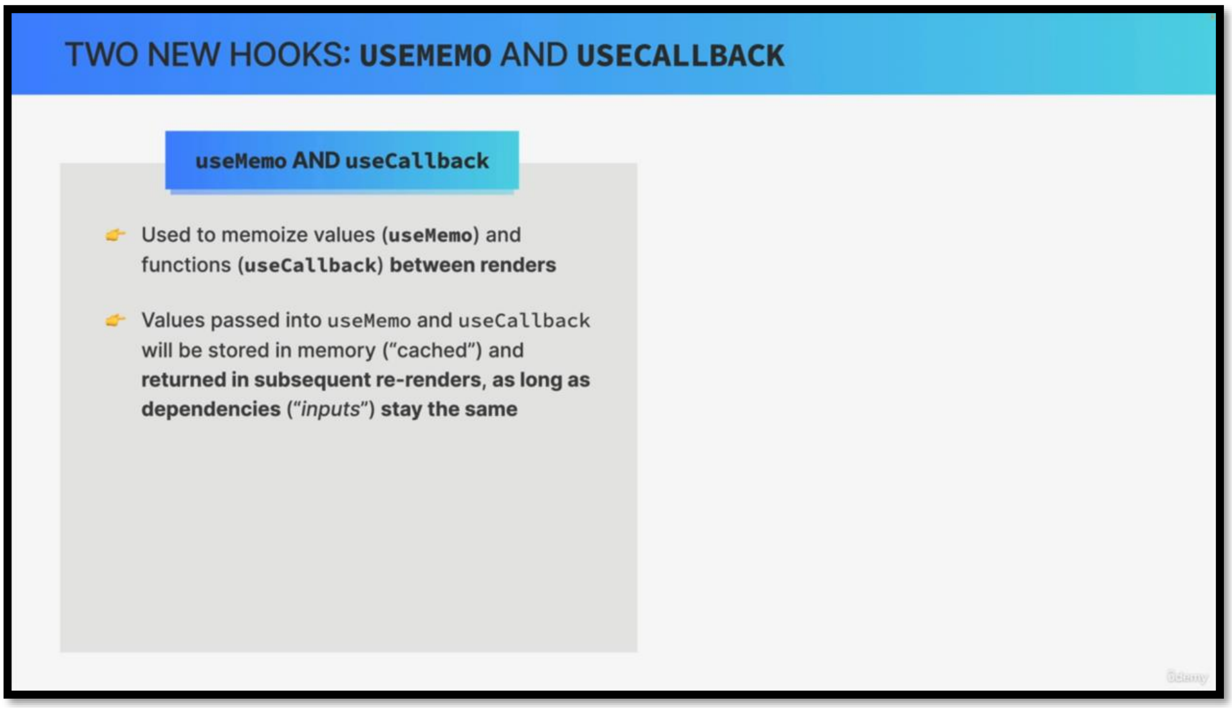
TWO NEW HOOKS: **USEMEMO AND USECALLBACK**

useMemo AND useCallback

- Used to memoize values (**useMemo**) and functions (**useCallback**) **between renders**

To do that, React gives us two more hooks i.e. UseMemo and useCallback.

So, we can use useMemo to memoize any value that we want to preserve between renders and useCallback to memoize functions between renders.



TWO NEW HOOKS: **USEMEMO AND USECALLBACK**

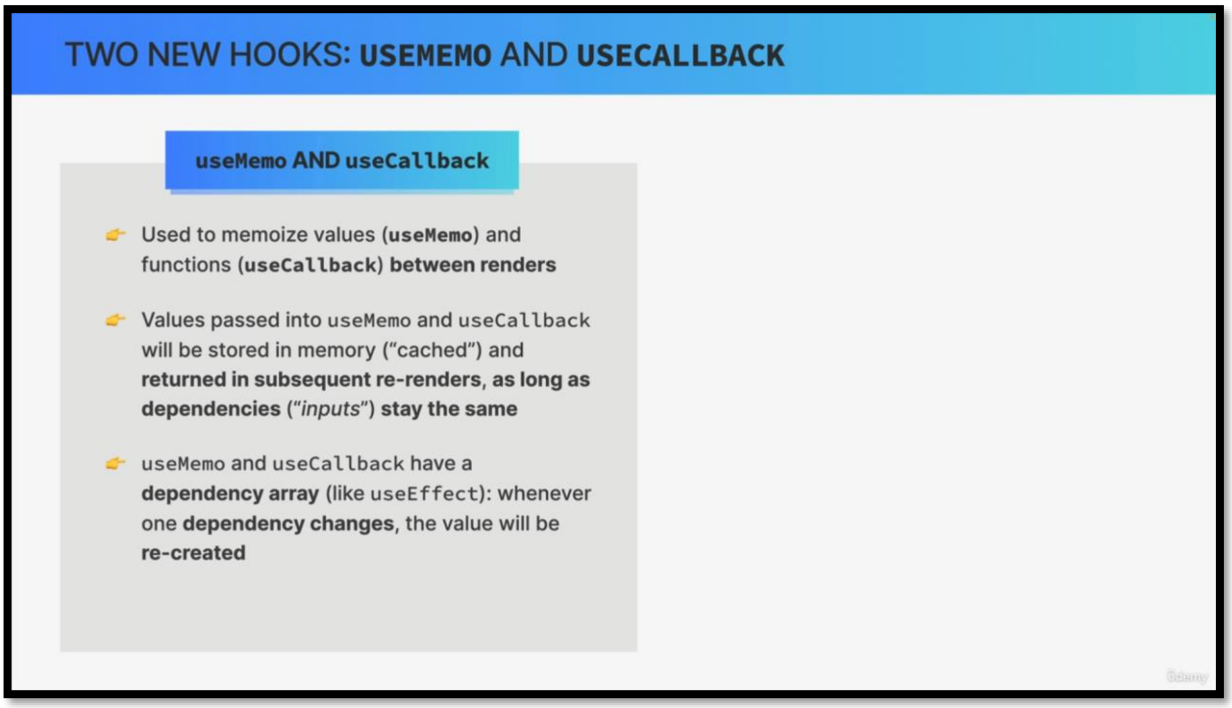
useMemo AND useCallback

- Used to memoize values (**useMemo**) and functions (**useCallback**) **between renders**
- Values passed into useMemo and useCallback will be stored in memory ("cached") and **returned in subsequent re-renders, as long as dependencies ("*inputs*") stay the same**

So whatever value that we pass into useMemo or useCallback will basically be stored in memory and that cached value will then be returned in future re-renders.

So it will be preserved across renders as long as the inputs stay the same.

Now in the case of useMemo and useCallback these inputs that we just mentioned are called dependencies.



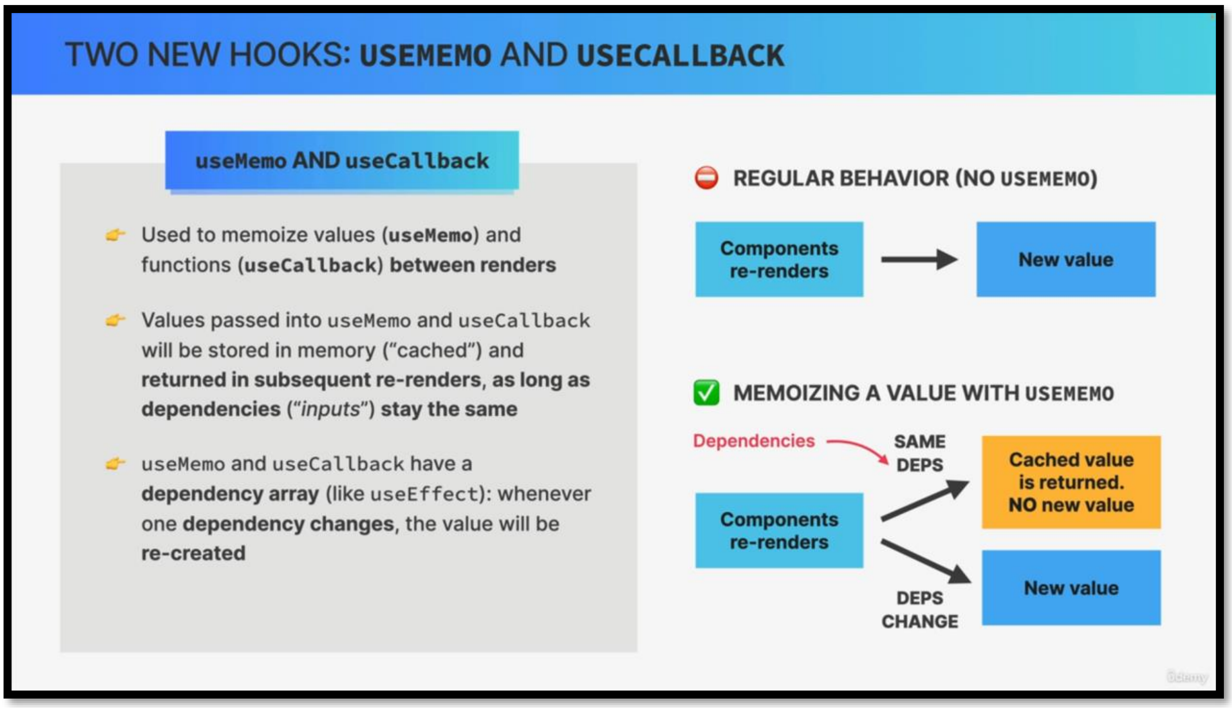
TWO NEW HOOKS: **USEMEMO AND USECALLBACK**

useMemo AND useCallback

- Used to memoize values (**useMemo**) and functions (**useCallback**) **between renders**
- Values passed into useMemo and useCallback will be stored in memory ("cached") and **returned in subsequent re-renders, as long as dependencies** ("*inputs*") **stay the same**
- useMemo and useCallback have a **dependency array** (like useEffect): whenever one **dependency changes**, the value will be **re-created**

So just like the useEffect hook, useMemo and useCallback also have a dependency array. Whenever one of the dependencies change the value will no longer be returned from the cache but will instead be recreated.

So this is very similar to the memo function where a component gets recreated whenever the props change. It's just a different thing that we're memorizing here and a different way of specifying the inputs but the idea is the same.



TWO NEW HOOKS: USEMEMO AND USECALLBACK

useMemo AND useCallback

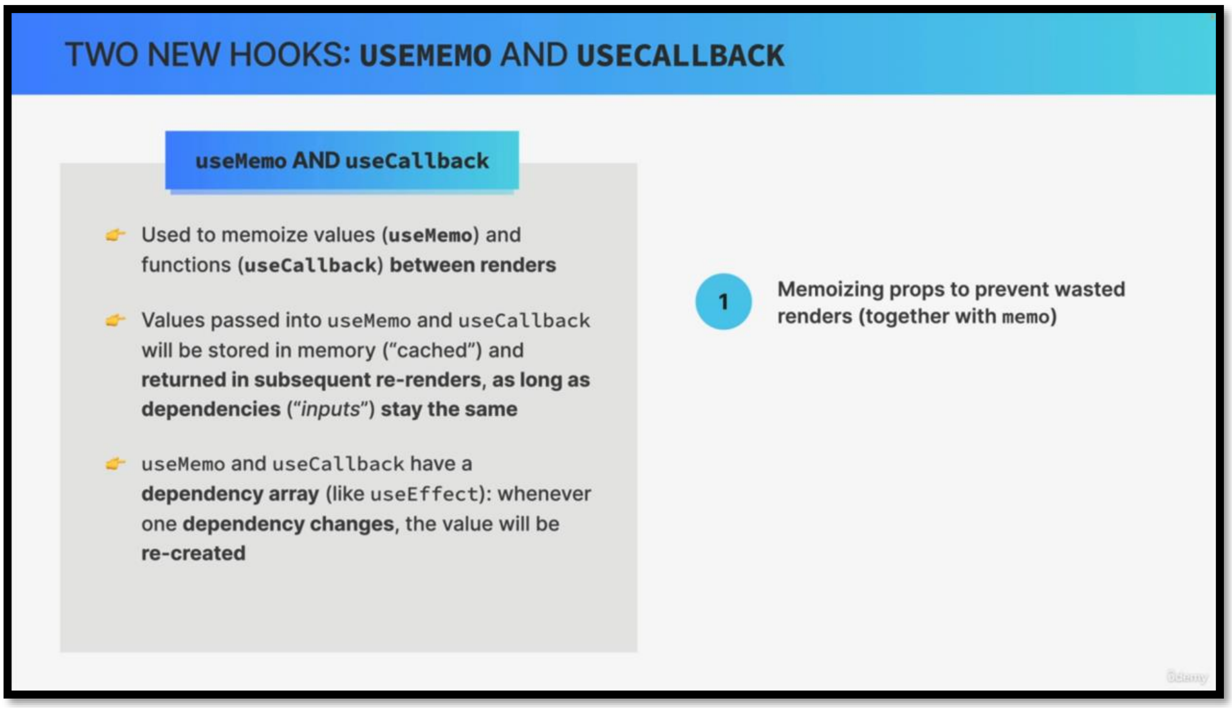
- Used to memoize values (**useMemo**) and functions (**useCallback**) **between renders**
- Values passed into useMemo and useCallback will be stored in memory ("cached") and **returned in subsequent re-renders, as long as dependencies** ("*inputs*") **stay the same**
- useMemo and useCallback have a **dependency array** (like useEffect): whenever one **dependency changes**, the value will be **re-created**

REGULAR BEHAVIOR (NO USEMEMO)
Components re-renders
New value
MEMOIZING A VALUE WITH USEMEMO
Dependencies
SAME DEPS
Cached value is returned. NO new value
Components re-renders
DEPS CHANGE
New value

So the regular behavior in React when we do not memoize a certain value is of course that a new value is created whenever the component re-renders.

On the other hand, when we do memoize the value then no new value is created on re-render and the cached value is returned instead and so like this, the value will stay exactly the same. So, it will be stable across renders.

However, this is only true if the dependencies that we specify in the dependency array don't change. If they do change, then a new value is actually created as if the memorization has never happened.



TWO NEW HOOKS: **USEMEMO AND USECALLBACK**

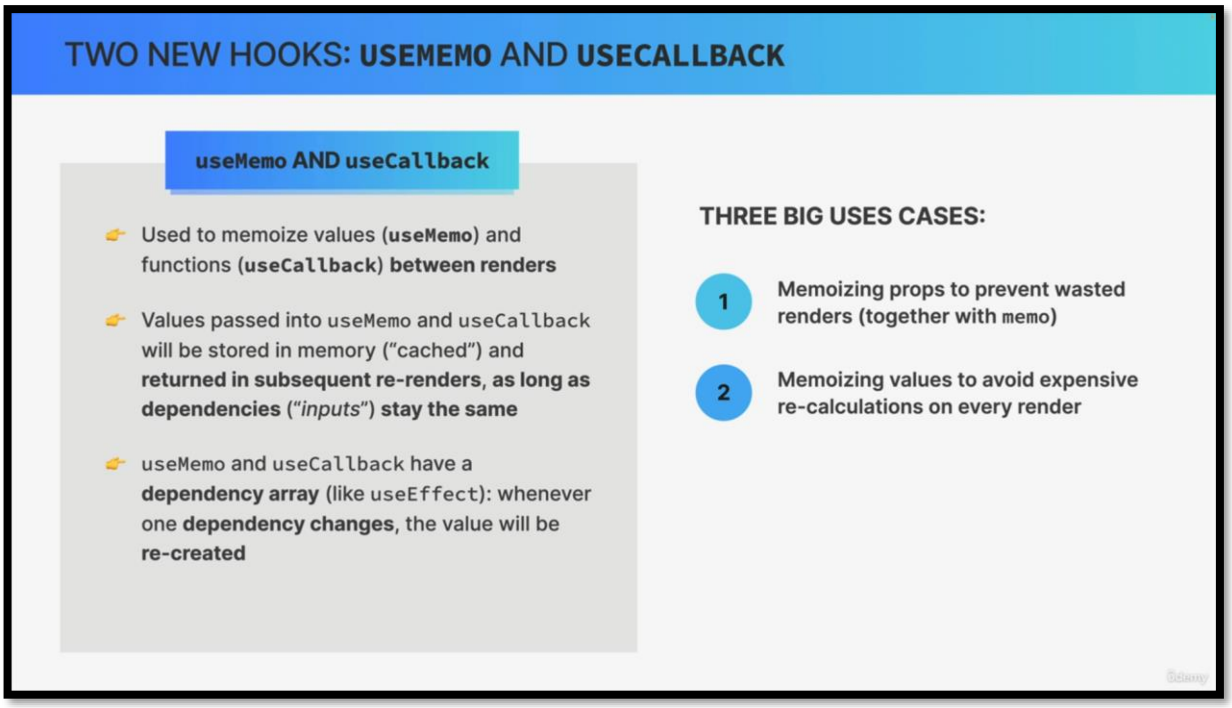
useMemo AND useCallback

- Used to memoize values (**useMemo**) and functions (**useCallback**) **between renders**
- Values passed into useMemo and useCallback will be stored in memory ("cached") and **returned in subsequent re-renders, as long as dependencies ("*inputs*") stay the same**
- useMemo and useCallback have a **dependency array** (like useEffect): whenever one **dependency changes**, the value will be **re-created**

1 **Memoizing props to prevent wasted renders (together with memo)**

We need to make objects or functions stable in order to actually make the memo function work.

In other words, if props are objects or functions, we need to memoize these props in order to prevent wasted renders.



TWO NEW HOOKS: **USEMEMO AND USECALLBACK**

useMemo AND useCallback

- Used to memoize values (**useMemo**) and functions (**useCallback**) **between renders**
- Values passed into useMemo and useCallback will be stored in memory ("cached") and **returned in subsequent re-renders, as long as dependencies ("*inputs*") stay the same**
- useMemo and useCallback have a **dependency array** (like useEffect): whenever one **dependency changes**, the value will be **re-created**

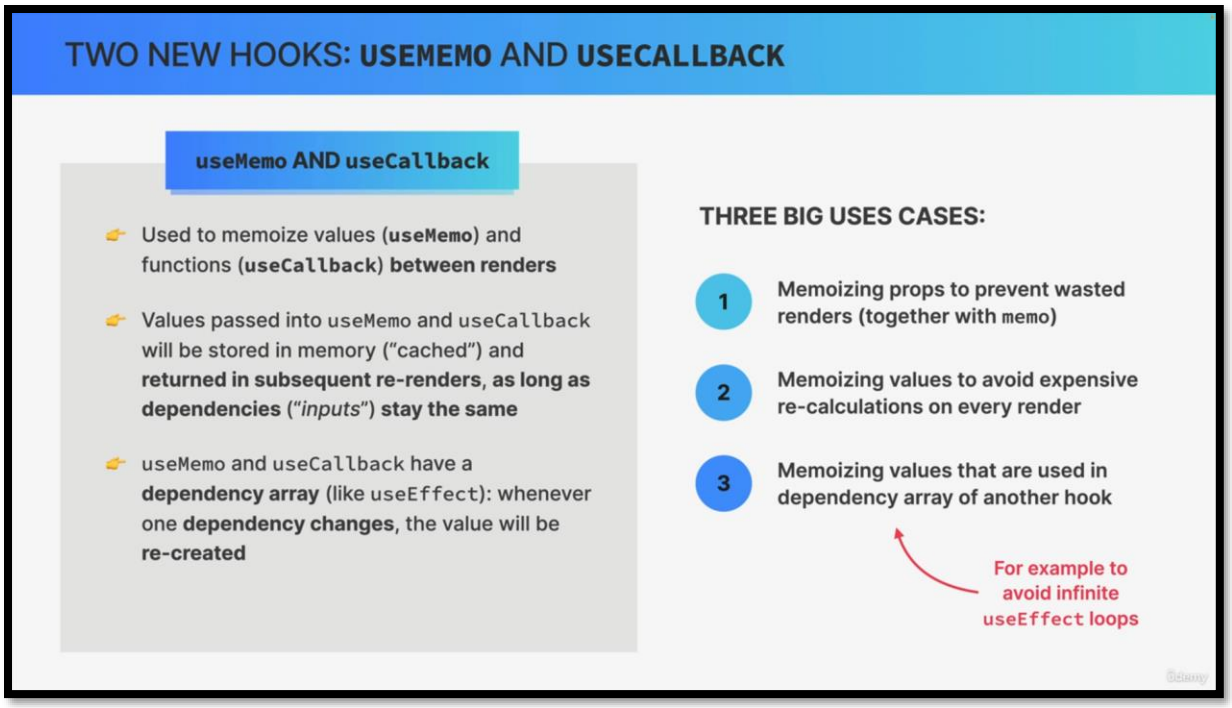
THREE BIG USES CASES:

1 **Memoizing props to prevent wasted renders (together with memo)**

2 **Memoizing values to avoid expensive re-calculations on every render**

The second use case is to avoid expensive recalculations on every render. For example, you might have a derived state that is calculated from an array of 100,000 items. Now, if your component re-renders all the time, then React needs to do this expensive calculation over and over again each time there is a render.

So, to fix this, you can simply preserve the results of that calculation across renders using useMemo and so then React doesn't have to calculate the same thing time and time again.



TWO NEW HOOKS: **USEMEMO AND USECALLBACK**

useMemo AND useCallback

- Used to memoize values (**useMemo**) and functions (**useCallback**) **between renders**
- Values passed into useMemo and useCallback will be stored in memory ("cached") and **returned in subsequent re-renders, as long as dependencies** ("*inputs*") **stay the same**
- useMemo and useCallback have a **dependency array** (like useEffect): whenever one **dependency changes**, the value will be **re-created**

THREE BIG USES CASES:

1. **Memoizing props to prevent wasted renders (together with memo)**
2. **Memoizing values to avoid expensive re-calculations on every render**
3. **Memoizing values that are used in dependency array of another hook**

For example to avoid infinite useEffect loops

Finally, another use case is memorizing values that are used in the dependency array of other hooks, for example, in order to avoid infinite useEffect loops.

Now, just like with the memo function it's important to not overuse these hooks. So, you should probably only use them for one of these three use cases and not start memorizing every single object and function everywhere.