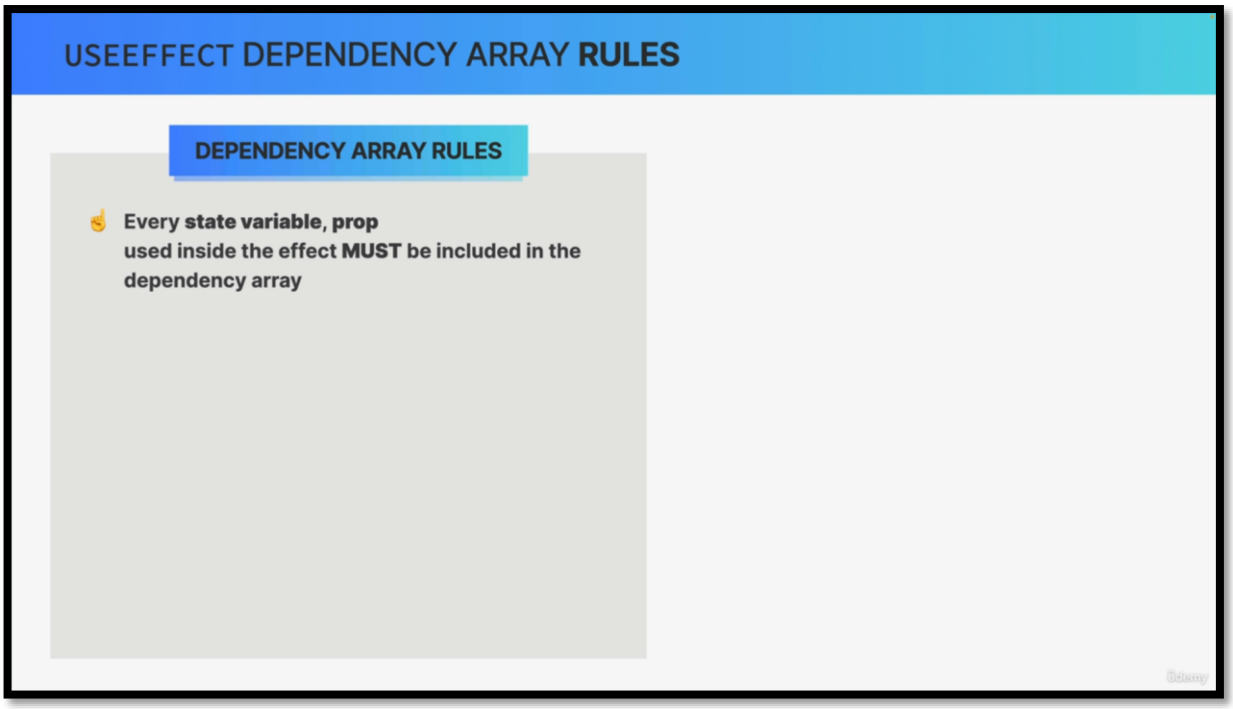


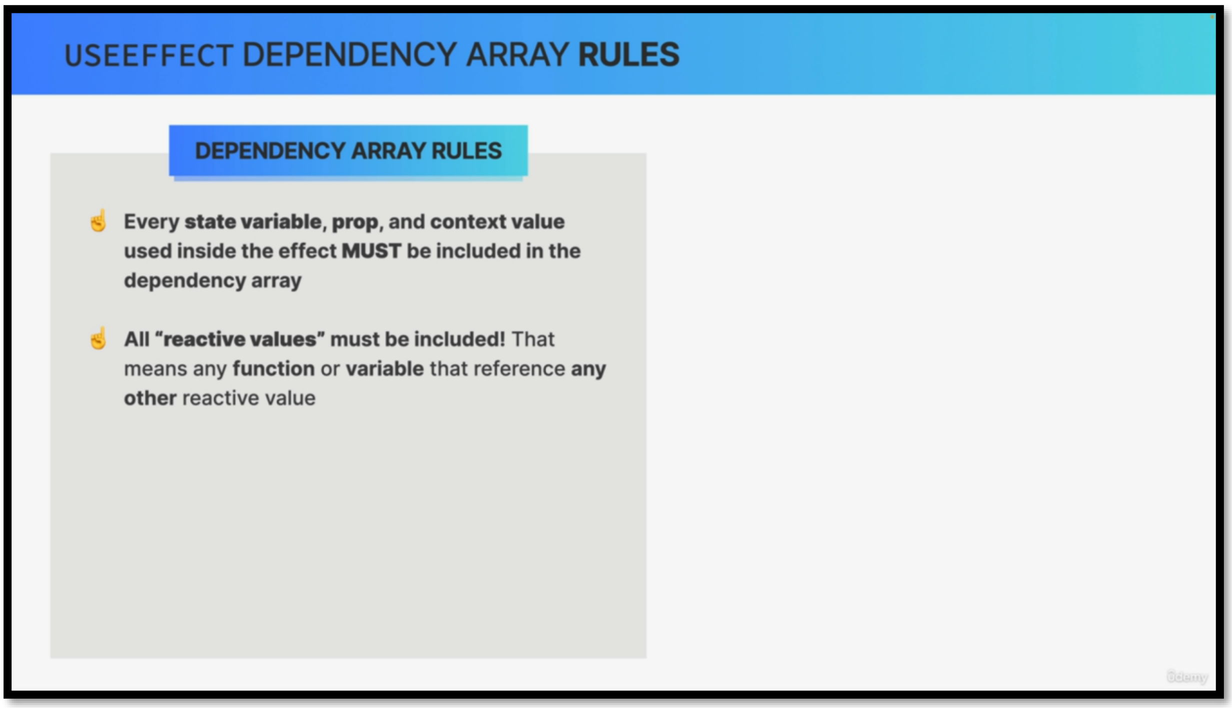
USEEFFECT DEPENDENCY ARRAY **RULES**

DEPENDENCY ARRAY RULES

- Every **state variable, prop** used inside the effect **MUST** be included in the dependency array

The first rule that we learned when we first talked about useEffect was that every single state variable and prop that's being used in the effect must be included in the dependency array.

However, this rule is not 100% complete because of two things.

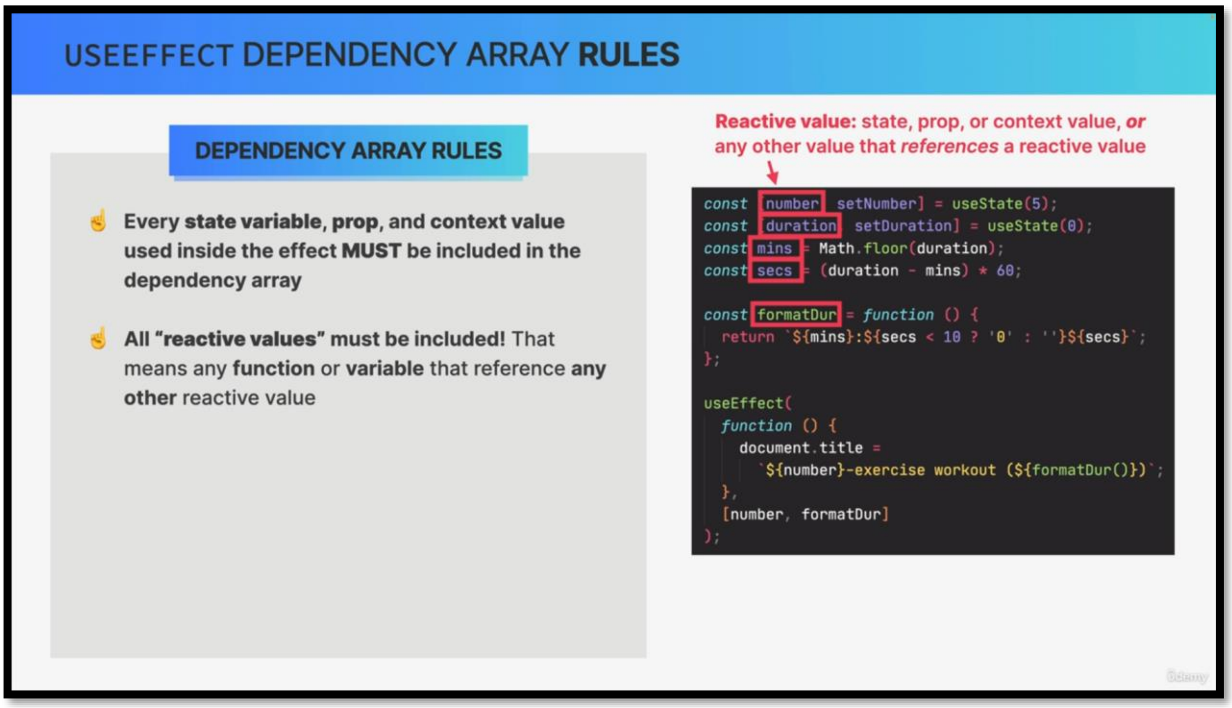


USEEFFECT DEPENDENCY ARRAY **RULES**

DEPENDENCY ARRAY RULES

- Every **state variable, prop,** and **context value** used inside the effect **MUST** be included in the dependency array
- **All "reactive values" must be included!** That means any **function** or **variable** that reference **any other** reactive value

First, any context value that a component is subscribed to must also be included in the dependency array and second, actually we must include all so-called reactive values in the dependency array as well.



USEEFFECT DEPENDENCY ARRAY **RULES**

DEPENDENCY ARRAY RULES

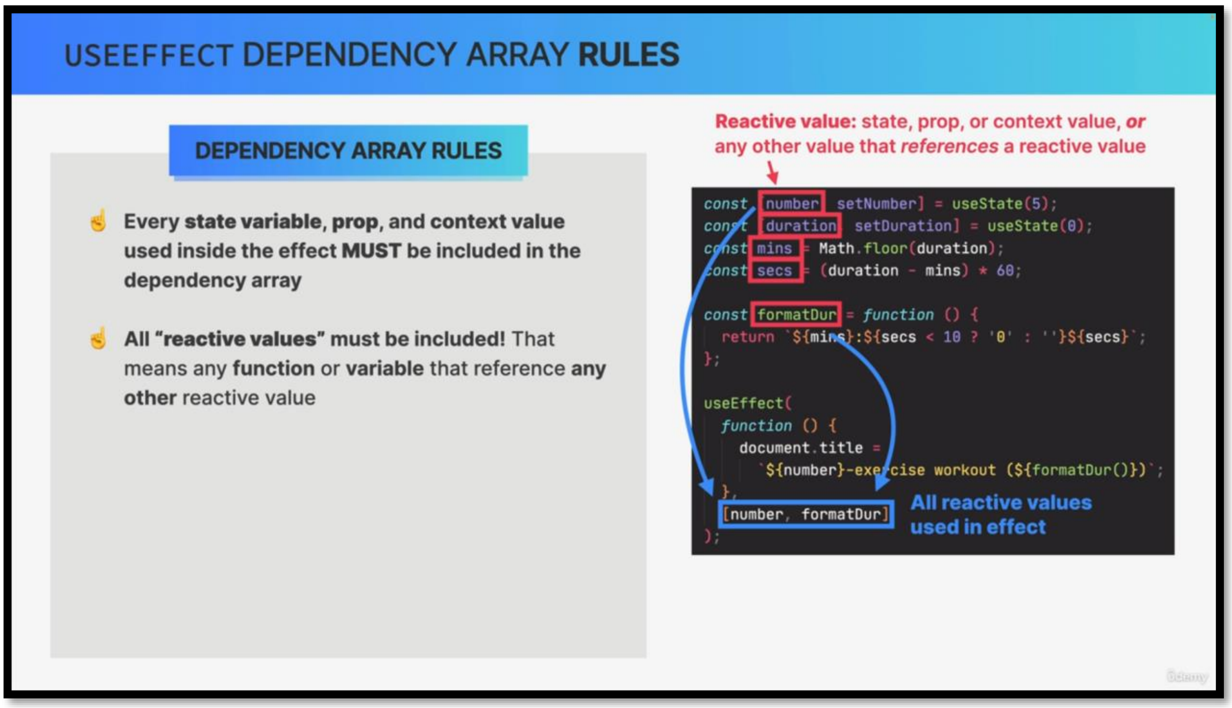
- ☝️ Every **state variable, prop,** and **context value** used inside the effect **MUST** be included in the dependency array
- ☝️ **All "reactive values"** must be included! That means any **function** or **variable** that reference **any other** reactive value

Reactive value: state, prop, or context value, *or* any other value that *references* a reactive value

```
const [number, setNumber] = useState(5);
const [duration, setDuration] = useState(0);
const mins = Math.floor(duration);
const secs = (duration - mins) * 60;

const formatDur = function () {
  return `${mins}:${secs < 10 ? '0' : ''}${secs}`;
};

useEffect(
  function () {
    document.title =
      `${number}-exercise workout (${formatDur()})`;
  },
  [number, formatDur]
);
```

Well, a reactive value is any value that is either state, a prop, a context value, or any other value that itself references one of the reactive values.

So, basically all values that are somehow connected to state, props, or context are reactive values.



USEEFFECT DEPENDENCY ARRAY **RULES**

DEPENDENCY ARRAY RULES

- ☝️ Every **state variable, prop,** and **context value** used inside the effect **MUST** be included in the dependency array
- ☝️ **All "reactive values"** must be included! That means any **function** or **variable** that reference **any other** reactive value

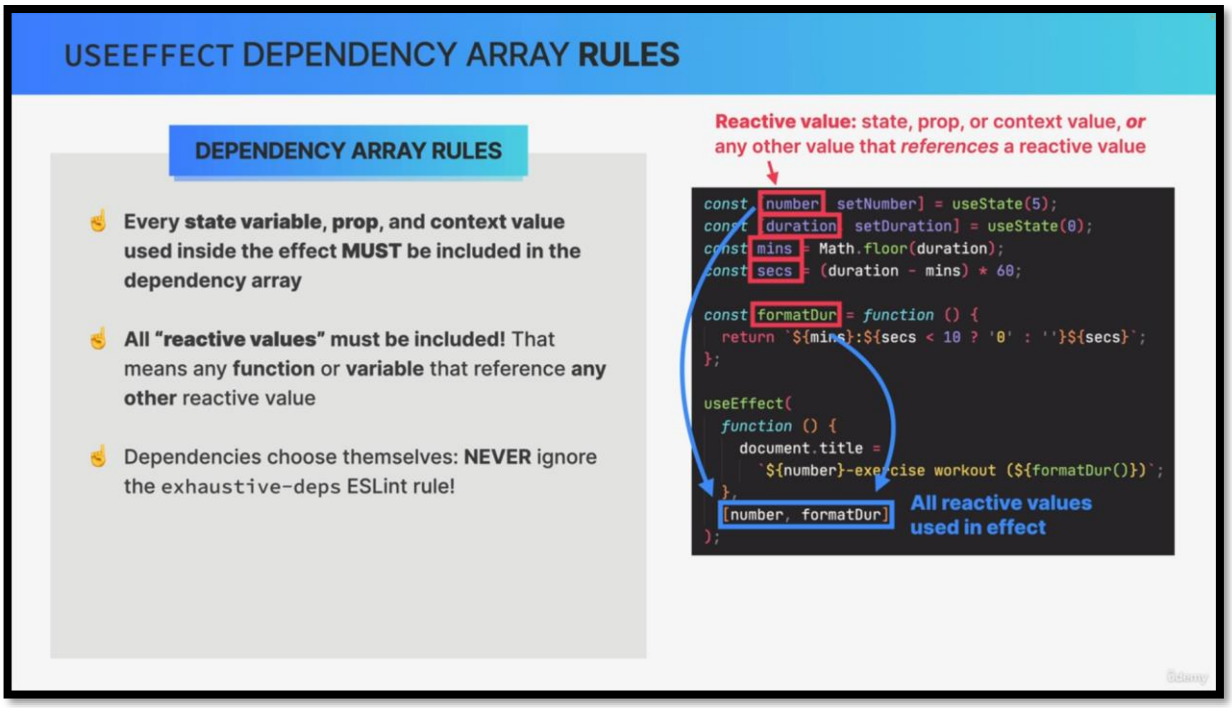
Reactive value: state, prop, or context value, *or* any other value that *references* a reactive value

```
const [number, setNumber] = useState(5);
const [duration, setDuration] = useState(0);
const mins = Math.floor(duration);
const secs = (duration - mins) * 60;

const formatDur = function () {
  return `${mins}:${secs < 10 ? '0' : ''}${secs}`;
};

useEffect(
  function () {
    document.title =
      `${number}-exercise workout (${formatDur()})`;
  },
  [number, formatDur]
);
```

All reactive values used in effect

Now, going back to our dependency array, this means that every single reactive value must be included, which again, includes any functions or variables that reference any other reactive value.

The goal of this is to avoid so-called stale closures that would otherwise occur inside the effect function.



USEEFFECT DEPENDENCY ARRAY **RULES**

DEPENDENCY ARRAY RULES

- ☝️ Every **state variable**, **prop**, and **context value** used inside the effect **MUST** be included in the dependency array
- ☝️ All **"reactive values"** must be included! That means any **function** or **variable** that reference **any other** reactive value
- ☝️ Dependencies choose themselves: **NEVER** ignore the `exhaustive-deps` ESLint rule!

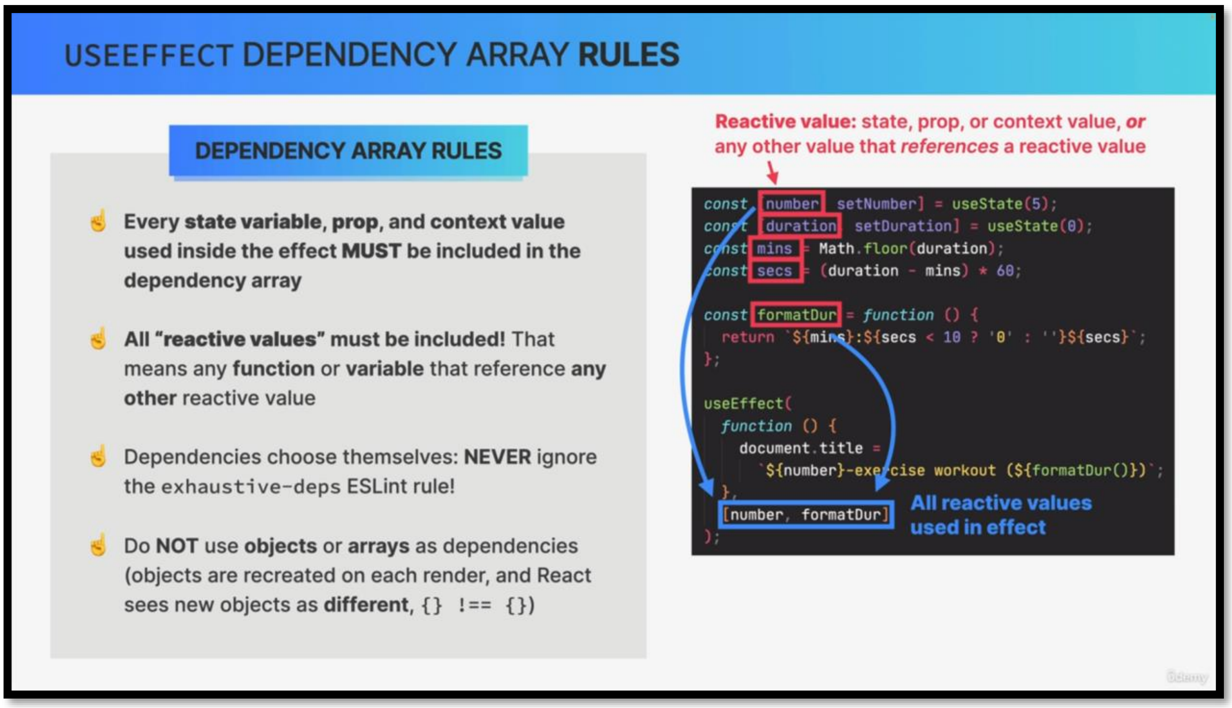
Reactive value: state, prop, or context value, ***or*** any other value that *references* a reactive value

```
const [number, setNumber] = useState(5);
const [duration, setDuration] = useState(0);
const mins = Math.floor(duration);
const secs = (duration - mins) * 60;

const formatDur = function () {
  return `${mins}:${secs < 10 ? '0' : ''}${secs}`;
};

useEffect(
  function () {
    document.title =
      `${number}-exercise workout (${formatDur()})`;
  },
  [number, formatDur]
);
```

All reactive values used in effect

Now, these two rules that we just learned about are pretty strict and tell us exactly what must be included in the dependency array and so this means that basically the dependencies choose themselves.

Therefore, you should never, ever ignore that ESLint rule that we've seen time and time again warning us about missing dependencies and ensuring that our dependency arrays are correct at all times.



USEEFFECT DEPENDENCY ARRAY **RULES**

DEPENDENCY ARRAY RULES

- ☝️ Every **state variable**, **prop**, and **context value** used inside the effect **MUST** be included in the dependency array
- ☝️ **All "reactive values" must be included!** That means any **function** or **variable** that reference **any other** reactive value
- ☝️ Dependencies choose themselves: **NEVER** ignore the `exhaustive-deps` ESLint rule!
- ☝️ Do **NOT** use **objects** or **arrays** as dependencies (objects are recreated on each render, and React sees new objects as **different**, `{} !== {}`)

Reactive value: state, prop, or context value, ***or*** any other value that *references* a reactive value

```
const [number, setNumber] = useState(5);
const [duration, setDuration] = useState(0);
const mins = Math.floor(duration);
const secs = (duration - mins) * 60;

const formatDur = function () {
  return `${mins}:${secs < 10 ? '0' : ''}${secs}`;
};

useEffect(
  function () {
    document.title =
      `${number}-exercise workout (${formatDur()})`;
  },
  [number, formatDur]
);
```

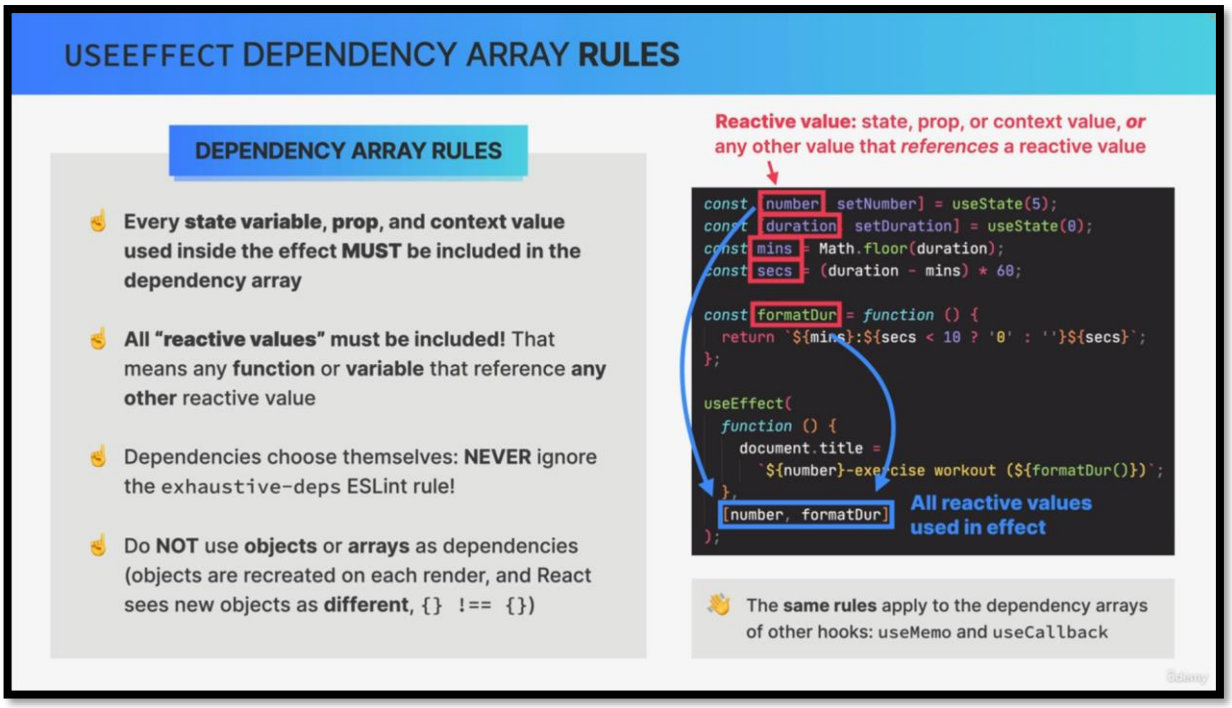
All reactive values used in effect

The final rule is that you should not use objects or arrays as dependencies, because when you do so, it seems to work just fine, but in reality, the effect will rerun on every single render.

The reason for that is that React checks if dependencies have changed between renders by simply comparing them using the triple equality operator.

However, objects in JavaScript will have a different reference each time that they are recreated.

Therefore, even if the content of an object stays the same after a render, React will still see the old and the new object as different and will therefore rerun the effect.



USEEFFECT DEPENDENCY ARRAY **RULES**

DEPENDENCY ARRAY RULES

- Every **state variable**, **prop**, and **context value** used inside the effect **MUST** be included in the dependency array
- **All "reactive values" must be included!** That means any **function** or **variable** that reference **any other** reactive value
- Dependencies choose themselves: **NEVER** ignore the `exhaustive-deps` ESLint rule!
- Do **NOT** use **objects** or **arrays** as dependencies (objects are recreated on each render, and React sees new objects as **different**, `{} !== {}`)

Reactive value: state, prop, or context value, ***or*** any other value that *references* a reactive value

```
const [number, setNumber] = useState(5);
const [duration, setDuration] = useState(0);
const mins = Math.floor(duration);
const secs = (duration - mins) * 60;

const formatDur = function () {
  return `${mins}:${secs < 10 ? '0' : ''}${secs}`;
};

useEffect(
  function () {
    document.title =
      `${number}-exercise workout (${formatDur()})`;
  },
  [number, formatDur]
);
```

All reactive values used in effect

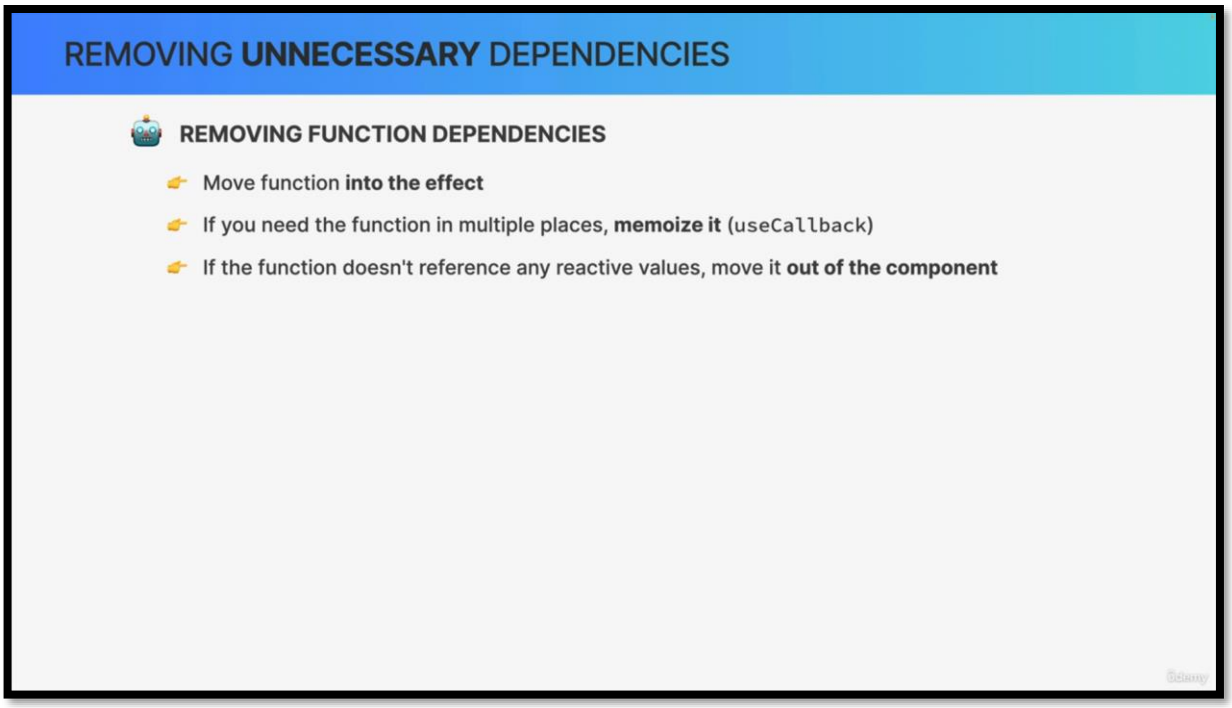
The **same rules** apply to the dependency arrays of other hooks: `useMemo` and `useCallback`

These rules that we just talked about work in the exact same way for the other hooks that also have dependency arrays, so useMemo and useCallback.

We must include all reactive values in the list of dependencies. React cost is not lying about dependencies.

However, in certain situations, including every single dependency in the array, can actually cost the effect to run way too often and introduce problems.

But again, the solution is not to omit one or more dependencies because that would be lying to React. Instead, we can use one of the strategies that can make some of the dependencies unnecessary, and so then we can remove those unnecessary dependencies from the array.



REMOVING **UNNECESSARY** DEPENDENCIES

REMOVING FUNCTION DEPENDENCIES

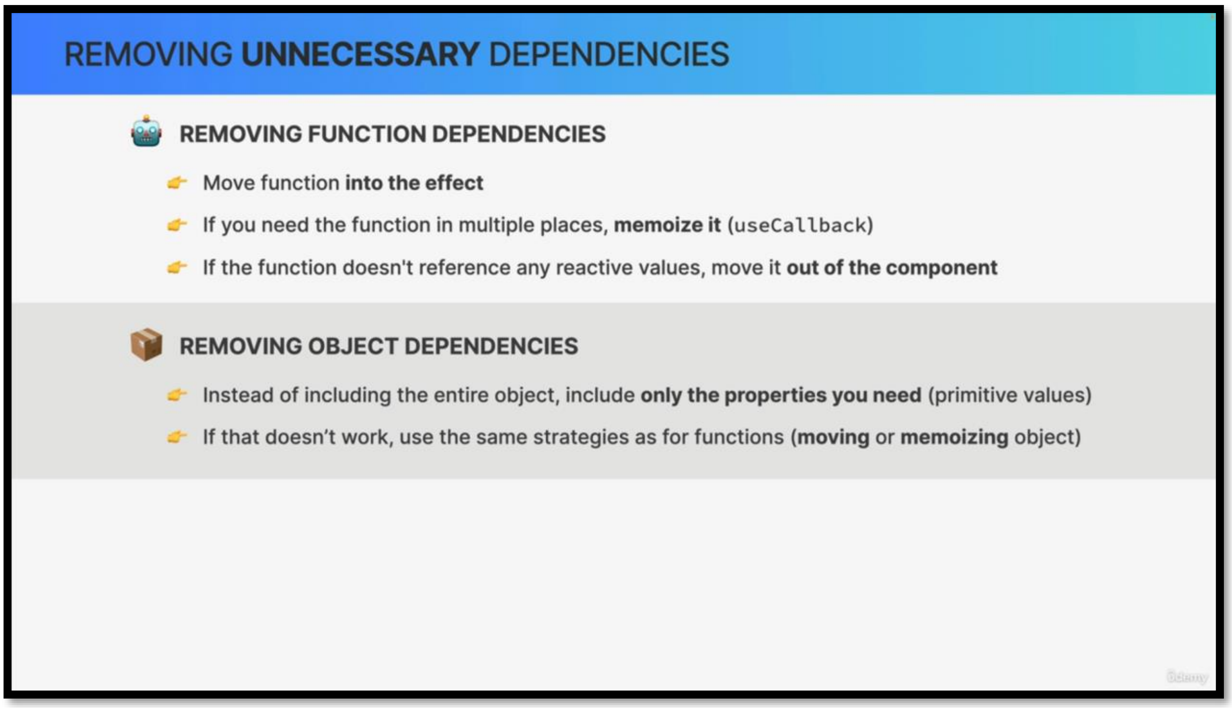
- Move function **into the effect**
- If you need the function in multiple places, **memoize it** (`useCallback`)
- If the function doesn't reference any reactive values, move it **out of the component**

So first, when you're dealing with helper functions as dependencies, the easiest way to remove the dependency is to just move the function right into the effect. Because if the function is inside the effect, then it's no longer a dependency of the effect.

However, if you can't do that, because you might need the same function in multiple places, you can try to memoize it with useCallback.

Also, if the function is not a reactive value itself, so if it doesn't use any reactive values inside the code, then you can just move it entirely out of the component because it's not really a dependency anyway.

So, by doing this, the function then doesn't need to be recreated on every render.



REMOVING **UNNECESSARY** DEPENDENCIES

REMOVING FUNCTION DEPENDENCIES

- Move function **into the effect**
- If you need the function in multiple places, **memoize it** (`useCallback`)
- If the function doesn't reference any reactive values, move it **out of the component**

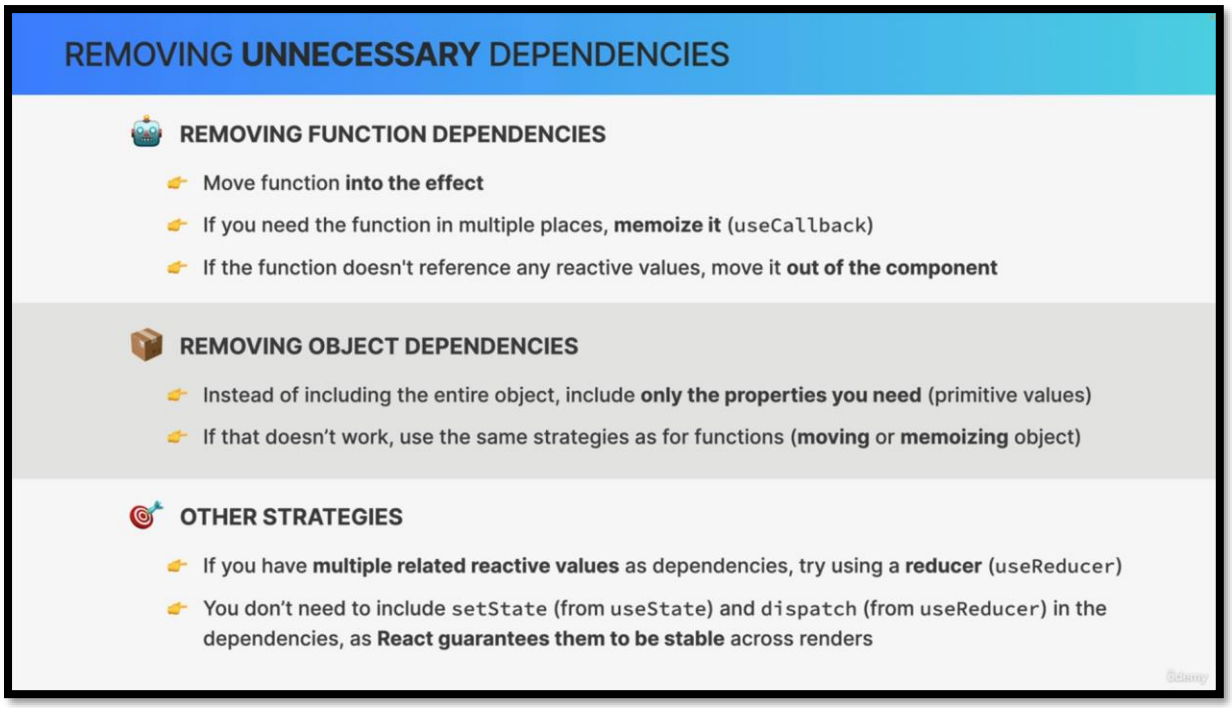
REMOVING OBJECT DEPENDENCIES

- Instead of including the entire object, include **only the properties you need** (primitive values)
- If that doesn't work, use the same strategies as for functions (**moving** or **memoizing** object)

Next up, if you want to use an object as a dependency, you can try to not include the entire object, but only the properties that you actually need inside the effect and you can do this as long as these properties are primitive values, like strings or numbers.

However, if for some reason that doesn't work for your specific situation, you can try one of the strategies that we mentioned for functions as they are quite similar.

Finally, we have two other strategies.



REMOVING **UNNECESSARY** DEPENDENCIES

🤖 REMOVING FUNCTION DEPENDENCIES

- 👉 Move function **into the effect**
- 👉 If you need the function in multiple places, **memoize it** (`useCallback`)
- 👉 If the function doesn't reference any reactive values, move it **out of the component**

📦 REMOVING OBJECT DEPENDENCIES

- 👉 Instead of including the entire object, include **only the properties you need** (primitive values)
- 👉 If that doesn't work, use the same strategies as for functions (**moving** or **memoizing** object)

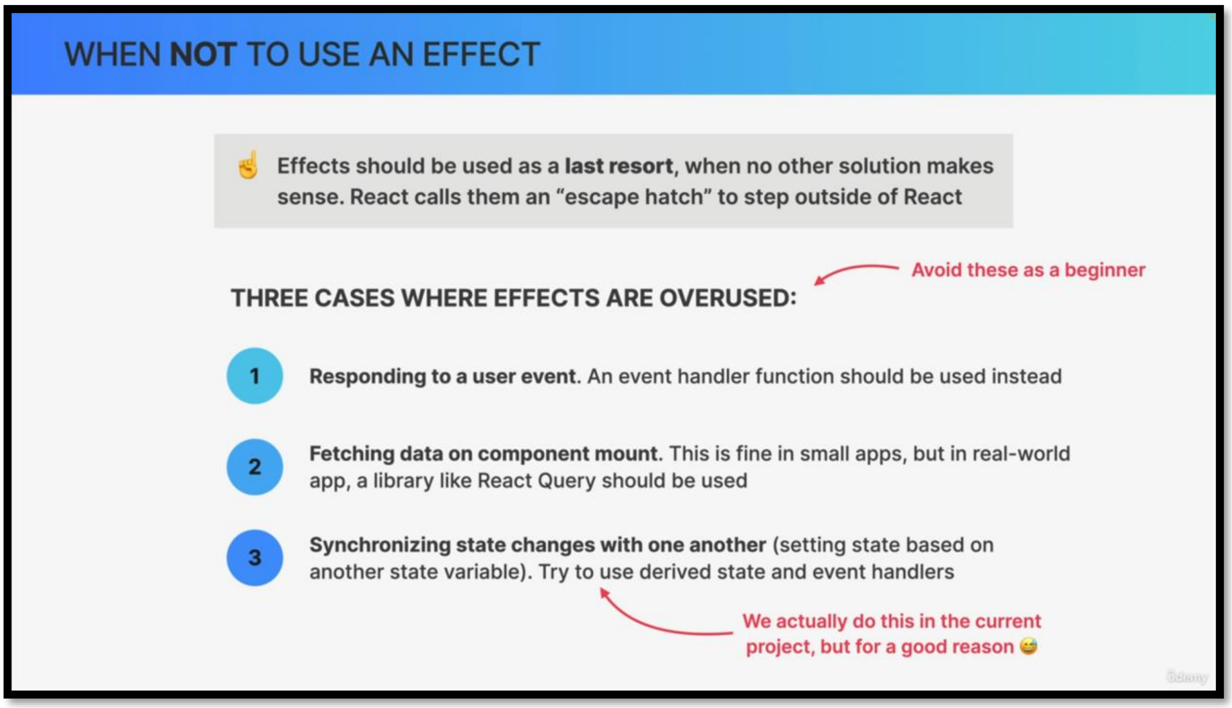
🎯 OTHER STRATEGIES

- 👉 If you have **multiple related reactive values** as dependencies, try using a **reducer** (`useReducer`)
- 👉 You don't need to include `setState` (from `useState`) and `dispatch` (from `useReducer`) in the dependencies, as **React guarantees them to be stable** across renders

First, if you find yourself in a situation where your dependency list includes multiple reactive values that are related to one another, you can try using a reducer with useReducer.

So sometimes a reducer can really be like a secret weapon that makes all your dependency problems completely go away.

Also, there is no need to include the setState function from useState or the dispatch function from useReducer in the dependency list becaus React guarantees that these are stable between renders.



WHEN **NOT** TO USE AN EFFECT

☝️ Effects should be used as a **last resort**, when no other solution makes sense. React calls them an "escape hatch" to step outside of React

THREE CASES WHERE EFFECTS ARE OVERUSED:

Avoid these as a beginner

1. **Responding to a user event**. An event handler function should be used instead
2. **Fetching data on component mount**. This is fine in small apps, but in real-world app, a library like React Query should be used
3. **Synchronizing state changes with one another** (setting state based on another state variable). Try to use derived state and event handlers

We actually do this in the current project, but for a good reason 😅