What to add and Not to add as Dependencies

In the previous lecture, we explored `useEffect()` dependencies.

You learned, that you should add "everything" you use in the effect function as a dependency - i.e. all state variables and functions you use in there.

That is correct, but there are a **few exceptions** you should be aware of:

- You **DON'T need to add state updating functions** (as we did in the last lecture with `setFormIsValid`): React guarantees that those functions never change, hence you don't need to add them as dependencies (you could though)
- You also **DON'T need to add "built-in" APIs or functions** like `fetch()`, `localStorage` etc (functions and features built-into the browser and hence available globally): These browser APIs / global functions are not related to the React component render cycle and they also never change
- You also **DON'T need to add variables or functions** you might've **defined OUTSIDE of your components** (e.g. if you create a new helper function in a separate file): Such functions or variables also are not created inside of a component function and hence changing them won't affect your components (components won't be re-evaluated if such variables or functions change and vice-versa)

So long story short: You must add all "things" you use in your effect function **if those "things" could change because your component (or some parent component) re-rendered.** That's why variables or state defined in component functions, props or functions defined in component functions have to be added as dependencies!

Here's a made-up dummy example to further clarify the above-mentioned scenarios:

```
1. import { useEffect, useState } from 'react';
2.
3. let myTimer;
4.
5. const MyComponent = (props) => {
6.   const [timerIsActive, setTimerIsActive] = useState(false);
7.
8.   const { timerDuration } = props; // using destructuring to pull out
   specific props values
9.
10.  useEffect(() => {
11.    if (!timerIsActive) {
12.      setTimerIsActive(true);
13.      myTimer = setTimeout(() => {
14.        setTimerIsActive(false);
15.      }, timerDuration);
16.    }
17.  }, [timerIsActive, timerDuration]);
18. };
```

In this example:

- `timerIsActive` is **added as a dependency** because it's component state that may change when the component changes (e.g. because the state was updated)
- `timerDuration` is **added as a dependency** because it's a prop value of that component - so it may change if a parent component changes that value (causing this MyComponent component to re-render as well)
- `setTimerIsActive` is **NOT added as a dependency** because it's that **exception**: State updating functions could be added but don't have to be added since React guarantees that the functions themselves never change
- `myTimer` is **NOT added as a dependency** because it's **not a component-internal variable** (i.e. not some state or a prop value) - it's defined outside of the component and changing it (no matter where) **wouldn't cause the component to be re-evaluated**
- `setTimeout` is **NOT added as a dependency** because it's **a built-in API** (built-into the browser) - it's independent from React and your components, it doesn't change