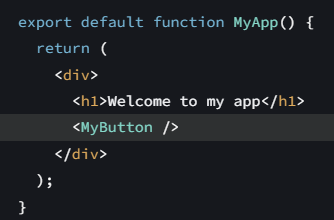
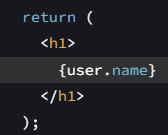
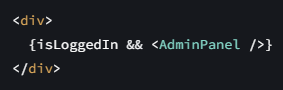
# **Learn React from documentation**

# **Quick Start**

**Contents:**

* How to create and nest components
* How to add markup and styles
* How to display data
* How to render conditions and lists
* How to respond to events and update the screen
* How to share data between components
* **Creating and Nesting Components**
  + React components are JavaScript functions that return markup.
  + Nesting

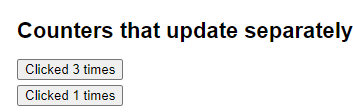
Nesting

* + The export default keywords specify the main component in the file.
* **Markup and Styling**
  + The markup syntax you’ve seen above is called JSX. It is optional, but most React projects use JSX for its convenience. All of the tools we recommend for local development support JSX out of the box.
  + If you have a lot of HTML to port to JSX, you can use an online converter.
  + In react we use className instead of class in regular HTML.
* **Displaying Data**
  + JSX lets you put markup into JavaScript. Curly braces let you “escape back” into JavaScript so that you can embed some variable from your code and display it to the user.
  + You can write normal JS inside the return value given you surround it with curly braces.
  + style={{}} is not a special syntax, but a regular {} object inside the style={ } JSX curly braces. You can use the style attribute when your styles depend on JavaScript variables.
* **Conditional Rendering**
  + In React, there is no special syntax for writing conditions. Instead, you’ll use the same techniques as you use when writing regular JavaScript code.
  + You can use the commonly used 3 type of conditional rendering statements
  + If Else ternary operator logical &&



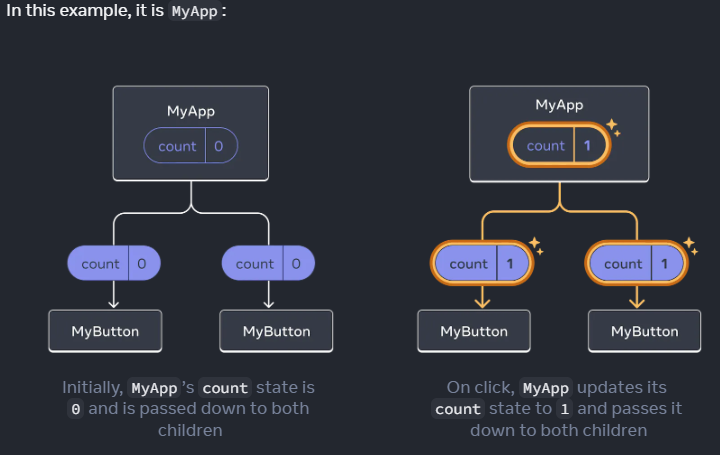
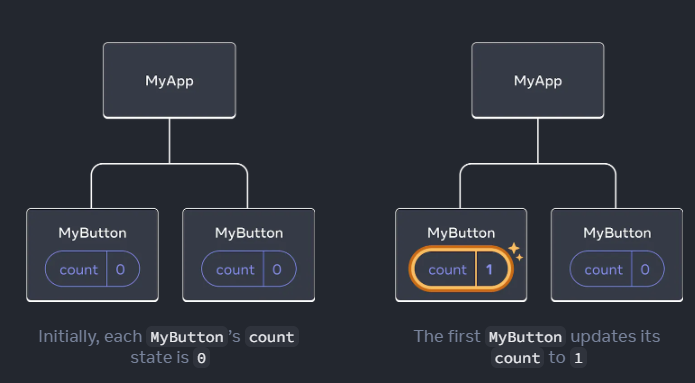
* + **Rendering lists:** You will rely on JavaScript features like for loop and the array map() function to render lists of components.
  + **Example:**
  + Notice how <li> has a key attribute. For each item in a list, you should pass a string or a number that uniquely identifies that item among its siblings. Usually, a key should be coming from your data, such as a database ID. React uses your keys to know what happened if you later insert, delete, or reorder the items.
* **Responding to Events and repaint**
  + You can respond to events by declaring event handler functions inside your components. These handler functions are passed to the onClick, onChange properties of the buttons or components in general.
  + Often, you’ll want your component to “remember” some information and display it.  
     For example, maybe you want to count the number of times a button is clicked.
  + To do this, add state to your component.  
     import {useState} from ‘react’;
  + Now you can declare a state variable inside your component.   
    function MyButton(){ const [count, setCount] = useState(0); //…}
  + You’ll get two things from useState: the current state (count), and the function that lets you update it (setCount). You can give them any names, but the convention is to write **[something, setSomething].**
  + The first time the button is displayed, count will be 0 because you passed 0 to useState(). When you want to change state, call setCount() and pass the new value to it. Clicking this button will increment the counter:

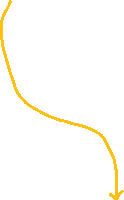
If you render the same component multiple times, each will get its own state. With a single state definition.

* + Functions starting with use are called Hooks
  + Hooks are more restrictive than other functions. You can only call Hooks at the top of your components (or other Hooks). If you want to use useState in a condition or a loop, extract a new component and put it there.
* **Share data between components: props**
  + **So in the above example you were able to manage state individually for each button ie segregated state.   
      
    *However, if you want to keep track of the total number of clicks in both button as one then you should define the state in that component which is a closest common parent to both component nodes.***

Common state for each button

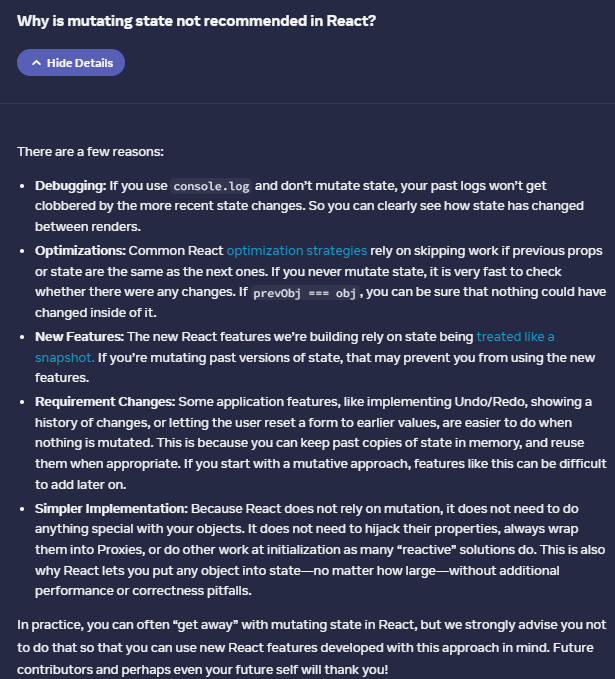
Separate state for each button

* + ****This in code is done through means of props. We send the state from parent to child component using props.



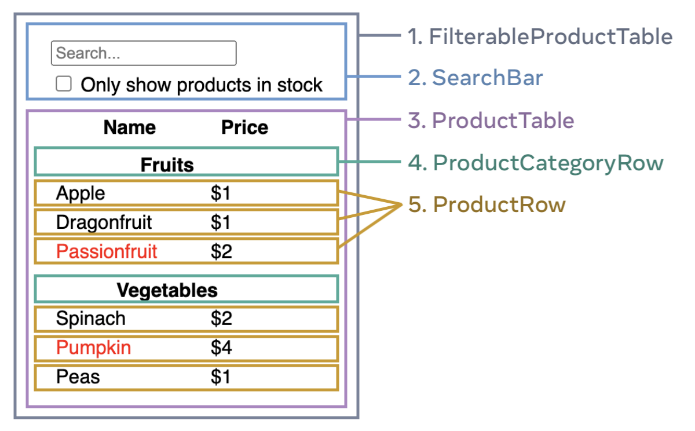
* + TICTACTOE [TUTORIAL](https://react.dev/learn/tutorial-tic-tac-toe)



* + In React, it’s conventional to use onSomething names for props which represent events and handleSomething for the function definitions which handle those events.
* **Why is immutability important?** 
  + Immutability makes it very cheap for components to compare whether their data has changed or not.
  + Immutability makes complex features much easier to implement.
  + Avoiding direct data mutation lets you keep previous versions of the data intact, and reuse them later.
* 
* Importance of Keys:
  + When a list is re-rendered, React takes each list item’s key and searches the previous list’s items for a matching key. **If the current list has a key that didn’t exist before, React creates a component. If the current list is missing a key that existed in the previous list, React destroys the previous component. If two keys match, the corresponding component is moved.**
  + Keys tell React about the identity of each component, which allows React to maintain state between re-renders. If a component’s key changes, the component will be destroyed and re-created with a new state.
  + key is a special and reserved property in React. When an element is created, React extracts the key property and stores the key directly on the returned element. Even though key may look like it is passed as props, React automatically uses key to decide which components to update. There’s no way for a component to ask what key its parent specified.

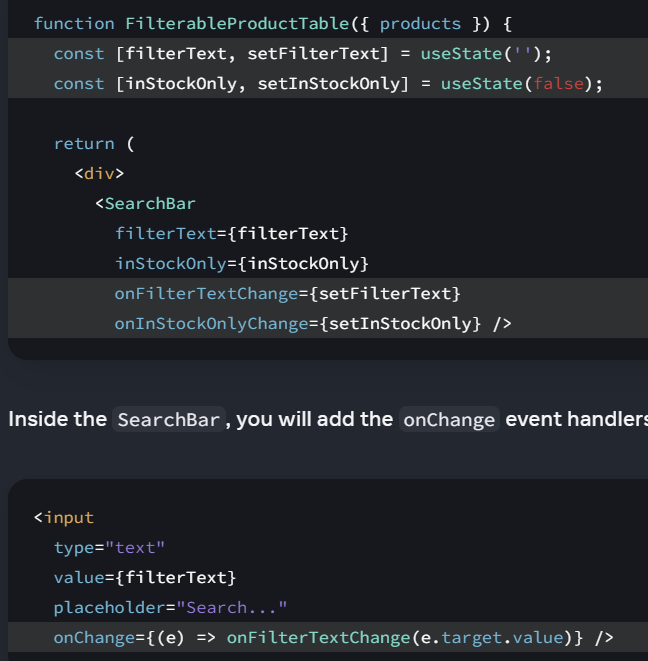
**Thinking in React**

* Overview
* Start with the mock-up

1. **Break the UI into a component hierarchy**
   1. Have the UI layout in front of you.
   2. Identify and break it into subparts could be done in following 3 ways:
      1. Like a developer would – based on [single responsibility rule.](https://en.wikipedia.org/wiki/Single_responsibility_principle)
      2. Maybe according to the CSS you think is different in each component, so based on the selectors.
      3. Design-layers.
   3. Now that you've identified the components in the mock-up, arrange them into a hierarchy. Components that appear within another component in the mock-up should appear as a child in the hierarchy:
      1. FilterableProductTable
         1. SearchBar
      2. ProductTable
         1. ProductCategoryRow
         2. ProductRow
2. **Build a static version in react**
   1. It’s time to implement your app. The most straightforward approach is to build a version that renders the UI from your data model without adding any interactivity… yet! It’s often easier to build the static version first and add interactivity later.
   2. Building a static version requires a lot of typing and no thinking, but adding interactivity requires a lot of thinking and not a lot of typing.
   3. **To build a static version of your app that renders your data model, you’ll want to build components that reuse other components and pass data using props. Props are a way of passing data from parent to child.**
   4. **The concept of state**: Don’t use state at all to build a static version. State is reserved only for interactivity, that is, data that changes over time. Sincethis is a static version of the app, youdon’t need it.
   5. Larger projects: **Build bottom up.**Smaller projects: **Build top down.** (in terms of hierarchy)
   6. After building your components, you’ll have a library of reusable components that render your data model. Because this is a static app, the components will only return JSX. The component at the top of the hierarchy (FilterableProductTable) will take your data model as a prop. This is called one-way data flow because the data flows down from the top-level component to the ones at the bottom of the tree.
3. **Find the minimal but complete representation of UI state**
   1. Think of state as the minimal set of changing data that your app needs to remember. The most important principle for structuring state is to keep it DRY (Don’t Repeat Yourself).
   2. Figure out the absolute minimal representation of the state your application needs and compute everything else on-demand.
   3. Now think of all of the pieces of data in this example application:
      1. The original list of products
      2. The search text the user has entered
      3. The value of the checkbox
      4. The filtered list of products
   4. Which of these are state? Identify the ones that are not:
   5. Does it remain unchanged over time? If so, it isn’t state. Is it passed in from a parent via props? If so, it isn’t state. Can you compute it based on existing state or props in your component? If so, it definitely isn’t state!
   6. What’s left is probably state.
   7. There are two types of “model” data in React: props and state. The two are very different:
   8. Props are like arguments you pass to a function. They let a parent component pass data to a child component and customize its appearance. For example, a Form can pass a color prop to a Button. State is like a component’s memory. It lets a component keep track of some information and change it in response to interactions. For example, a Button might keep track of isHovered state. Props and state are different, but they work together. A parent component will often keep some information in state (so that it can change it), and pass it down to child components as their props. It’s okay if the difference still feels fuzzy on the first read. It takes a bit of practice for it to really stick!
4. **Identify where your state should live**

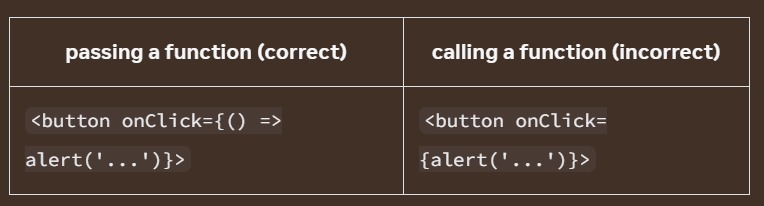
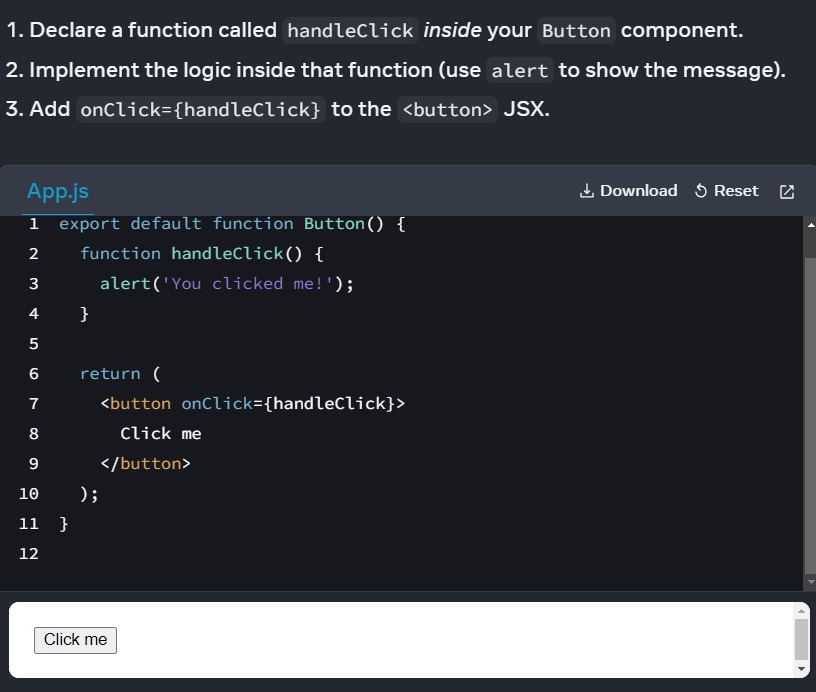
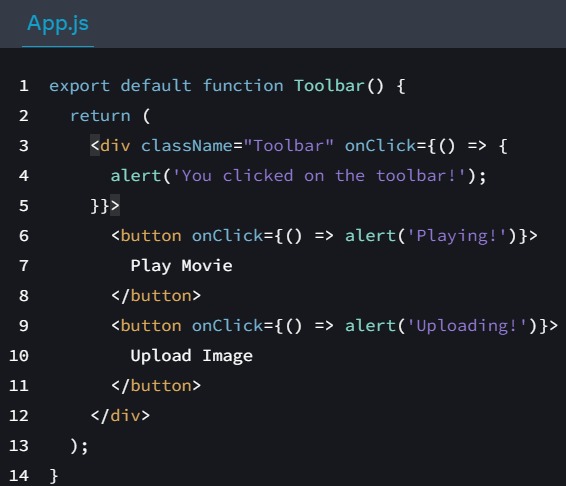
After identifying your app’s minimal state data, you need to identify which component is responsible for changing this state, or owns the state. Remember: React uses one-way data flow, passing data down the component hierarchy from parent to child component. It may not be immediately clear which component should own what state. For **each piece of state** in your application:

* 1. Identify every component that renders something based on that state.
  2. Find their closest common parent component—a component above them all in the hierarchy.
  3. Decide where the state should live:
     1. Often, you can **put the state directly into their common parent.**
     2. You can also put the state into **some component above their common parent. [Lifting the state]**
     3. If you can’t find a component where it makes sense to own the state, create a new component solely for holding the state and add it somewhere in the hierarchy above the common parent component. [**Wrapper component**]

1. **Add inverse data flow**
   1. to change the state according to user input, you will need to support data flowing the other way: the form components deep in the hierarchy need to update the state in FilterableProductTable.
   2. You want to make it such that, whenever the user changes the form inputs, the state updates to reflect those changes. The state is owned by FilterableProductTable, so only it can call setFilterText and setInStockOnly. To let SearchBar update the FilterableProductTable’s state, you need to pass these functions down to SearchBar:
   3. 

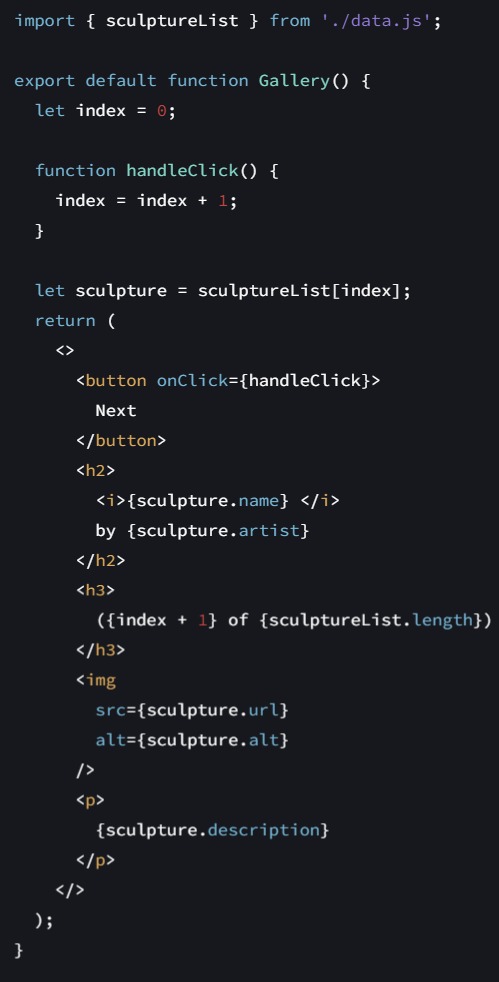
[More on Adding Interactivity](https://react.dev/learn/adding-interactivity)

Built-in components like <button> only support built-in browser events like onClick. However, you can also create your own components, and give their event handler props any application-specific names that you like.

* To add an event handler**, you will first define a function and then pass it as a prop to the appropriate JSX tag**. Functions passed to event handlers must be passed, not called.
* **Reading props in event handler:** Because event handlers are declared inside of a component, they have access to the component’s props.
* **Passing event handler as prop:** Often you’ll want the parent component to specify a child’s event handler. Consider buttons: depending on where you’re using a Button component, you might want to execute a different function—perhaps one plays a movie and another uploads an image. To do this, pass a prop the component receives from its parent as the event handler like shown in side.
* **Event propagation:** Event handlers will also catch events from any children your component might have. We say that an event “bubbles” or “propagates” up the tree: it starts with where the event happened, and then goes up the tree. If you click on either button, its onClick will run first, followed by the parent <div>’s onClick. So two messages will appear. If you click the toolbar itself, only the parent <div>’s onClick will run.   
    
  All events propagate in React except onScroll, which only works on the JSX tag you attach it to.
* If you want to prevent an event from reaching parent components, you need to call e.stopPropagation()

**State: A components Memory:**

## Why do we need a state? Why is a regular variable not enough?

Here’s a component that renders a sculpture image. Clicking the “Next” button should show the next sculpture by changing the index to 1, then 2, and so on. However, this won’t work (you can try it!):

**Why did this not work?**

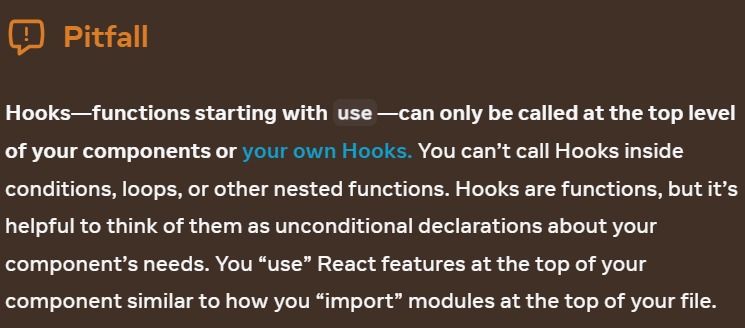
The handleClick event handler is updating a local variable, index. But two things prevent that change from being visible:

1. Local **variables don’t persist** between renders. When React renders this component a second time, it renders it from scratch—it doesn’t consider any changes to the local variables.
2. **Changes to local variables won’t trigger renders.** React doesn’t realize it needs to render the component again with the new data.

**To update a component with new data, two things need to happen:**

1. **Retain the data between renders.**
2. **Trigger React to render the component with new data (re-rendering).**

The **useState Hook** provides those two things. Every time your component renders, useState gives you an array containing two values:

1. A state variable to retain the data between renders.
2. A state setter function to update the variable and trigger React to render the component again.

You can have as many state variables of as many types as you like in one component.

[**Principles for structuring state**](https://react.dev/learn/choosing-the-state-structure)**🡨 Beautiful Article**

When you write a component that holds some state, you’ll have to make choices about how many state variables to use and what the shape of their data should be. While it’s possible to write correct programs even with a suboptimal state structure, there are a few principles that can guide you to make better choices:

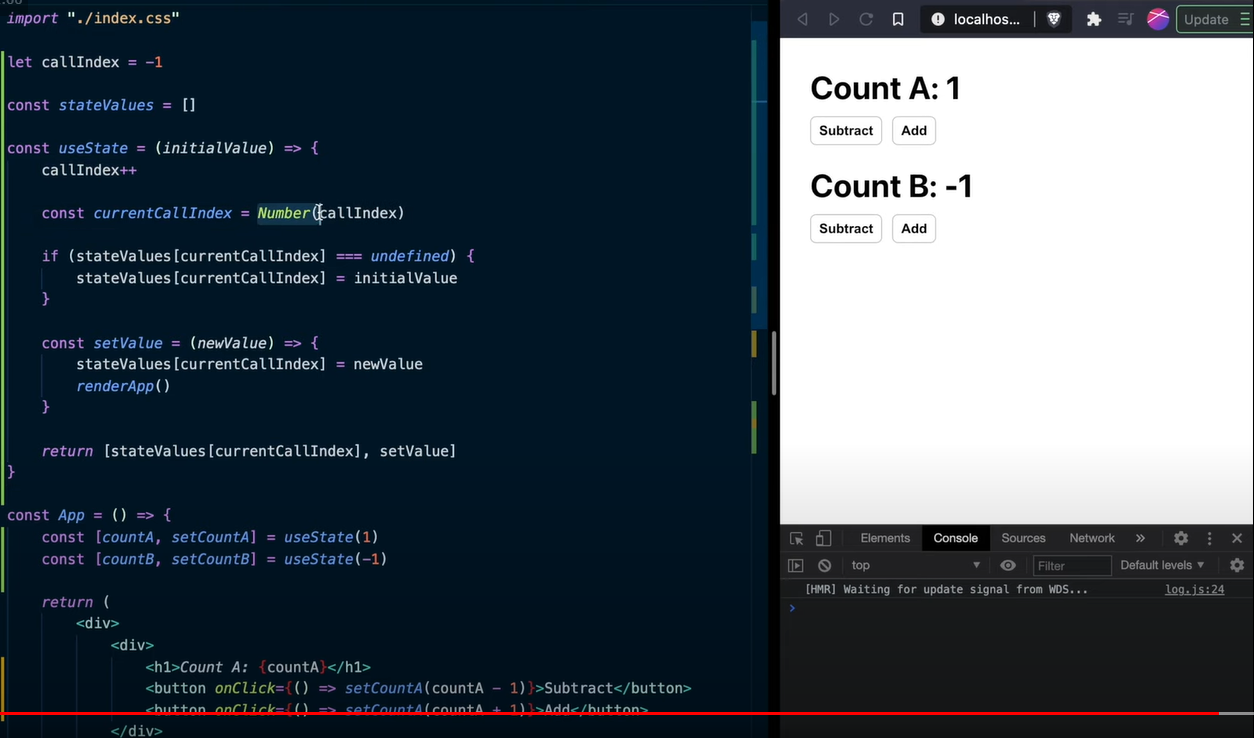
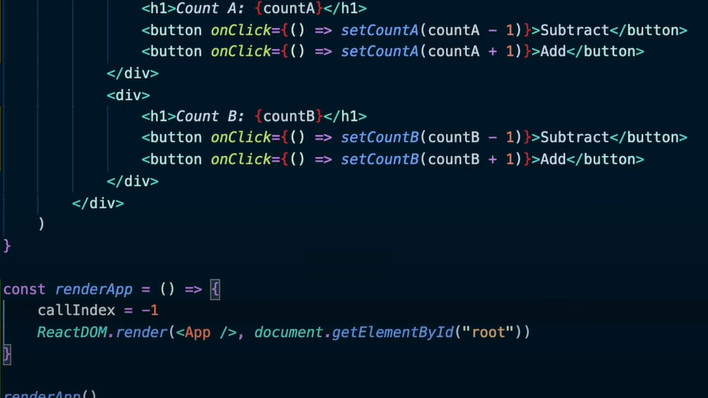
1. **Group related state.** If you always update two or more state variables at the same time, consider merging them into a single state variable.
2. **Avoid contradictions in state.** When the state is structured in a way that several pieces of state may contradict and “disagree” with each other, you leave room for mistakes. Try to avoid this.
3. **Avoid redundant state.** If you can calculate some information from the component’s props or its existing state variables during rendering, you should not put that information into that component’s state.
4. **Avoid duplication in state.** When the same data is duplicated between multiple state variables, or within nested objects, it is difficult to keep them in sync. Reduce duplication when you can.
5. **Avoid deeply nested state.** Deeply hierarchical state is not very convenient to update. When possible, prefer to structure state in a flat way.

#### **How does React know which state to return?**

#### [**FUNCTIONING OF REACT STATES IN ORDER**](https://medium.com/@ryardley/react-hooks-not-magic-just-arrays-cd4f1857236e)

Hooks rely on a stable call order on every render of the same component. This works well in practice because if you follow the rule above (“only call Hooks at the top level”), Hooks will always be called in the same order. Internally, React holds an array of state pairs for every component. It also maintains the current pair index, which is set to 0 before rendering. Each time you call useState, React gives you the next state pair and increments the index. You can read more about this mechanism in React Hooks: Not Magic, Just Arrays.

[SS video](https://youtu.be/_qkX3yAmgEw)

Scratch implementation of ReactHook.

[Since react maintains the state values based on the order of calls of hook that is why we cannot and should not use hooks in conditional statements or loops.](https://medium.com/@ryardley/react-hooks-not-magic-just-arrays-cd4f1857236e#:~:text=Why%20order%20is%20important)

Because we are dealing with a cursor pointing to a set of arrays, if you change the order of the calls within render, the cursor will not match up to the data and your use calls will not point to the correct data or handlers.

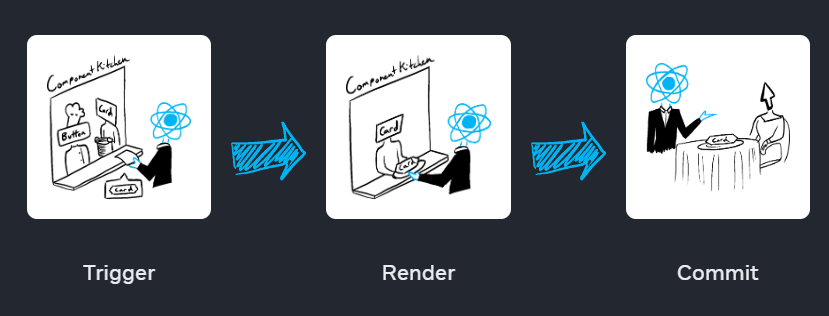
**Independent states**

**State is isolated and private.** State is local to a component instance on the screen. In other words, if you render the same component twice, each copy will have completely isolated state! Changing one of them will not affect the other.

**State Lifting/Sharing:**

What if you wanted 2 components to keep their states in sync? The right way to do it in React is to remove state from child components and add it to their closest shared parent. The next few pages will focus on organizing state of a single component, but we will return to this topic in Sharing State Between Components.

Render and Commit



1. Reasons to render:
   1. It’s the component’s initial render. Done by calling createRoot and then rendering it. 
   2. The component’s (or one of its ancestors’) state has been updated.
2. React renders your components:
   1. On initial render, React will call the root component after creating DOM nodes for the tags like section, h1, img etc.
   2. For subsequent renders, React will call the function component whose state update triggered the render.

**Recursive rendering**: if the updated component returns some other component, React will render that component next, and so on. The process will continue until there are no more nested components and React knows exactly what should be displayed on screen.

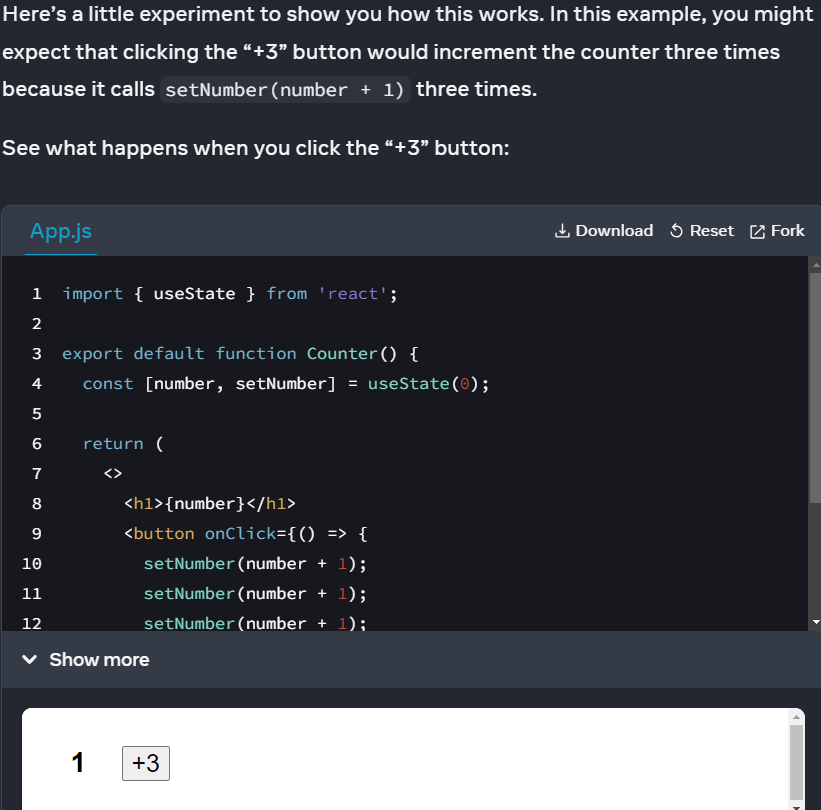
**Note that**[: React components must be a PURE FUNCTION](https://react.dev/learn/keeping-components-pure#:~:text=Why%20does%20React%20care%20about%20purity%3F)   
Rendering must always be a pure calculation: Same inputs, same output.

It minds its own business. It should not change any objects or variables that existed before rendering.

1. React commits changes to the DOM:  
   After **rendering (calling) your components**, React will modify the DOM.   
   For the initial render, React will use the appendChild() DOM API to put all the DOM nodes it has created on screen. For re-renders, React will apply the minimal necessary operations (calculated while rendering!) to make the DOM match the latest rendering output.
2. After rendering is done and React updated the DOM, the browser will repaint the screen. Although this process is known as “browser rendering”, we’ll refer to it as “painting” to avoid confusion throughout the docs.



State variables might look like regular JavaScript variables that you can read and write to. However, state behaves more like a snapshot[More details above in document: why is regular variable not enough?]. **As a component’s memory, state is not like a regular variable that disappears after your function returns.** State actually “lives” in React itself—as if on a shelf!—outside of your function.

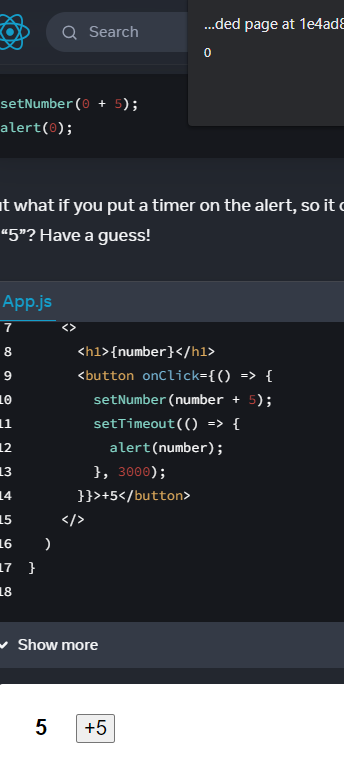
1. **How setting state triggers re-renders**For an interface to react to the event, you need to update the state.   
   When React re-renders a component:
   1. React **calls** your component pure function again.
   2. Your function returns a new JSX snapshot. The JSX you return from that function is like a snapshot of the UI in time.
2. **When and how state updates  
   PUZZLE 1**
3. React updates the screen to match the snapshot you’ve returned. Setting state only changes it for the next render. During the first render, number was 0. Here is what this button’s click handler tells React to do: [**Understand with Substitution method**]
4. setNumber(number + 1): number is 0 so setNumber(0 + 1).
5. React prepares to change number to 1 on the next render.



1. setNumber(number + 1): number is 0 so setNumber(0 + 1).
2. React prepares to change number to 1 on the next render.
3. setNumber(number + 1): number is 0 so setNumber(0 + 1).
4. React prepares to change number to 1 on the next render.

**Even though you called setNumber(number + 1) three times, in this render’s event handler number is always 0, so you set the state to 1 three times**. This is why, after your event handler finishes, React re-renders the component with number equal to 1 rather than 3.

1. **Why state does not update immediately after you set it**

**PUZZLE 2**

Surprised? If you use the substitution method, you can see the “snapshot” of the state passed to the alert.   
The state stored in React may have changed by the time the alert runs, but **it was scheduled using a snapshot of the state at the time the user interacted with it!**

**A state variable’s** **value never changes within a render**, even if its event handler’s code is asynchronous. Inside that render’s onClick, the value of number continues to be 0 even after setNumber(number + 5) was called. **Its value was “fixed” when React “took the snapshot” of the UI by calling your component.**

1. **How event handlers access a “snapshot” of the state**React keeps the state values “fixed” within one render’s event handlers. You don’t need to worry whether the state has changed while the code is running. But what if you wanted to read the latest state before a re-render**? You’ll want to use a state updater function COVERED IN NEXT SECTION ABOUT QUEUEING OF STATES.**

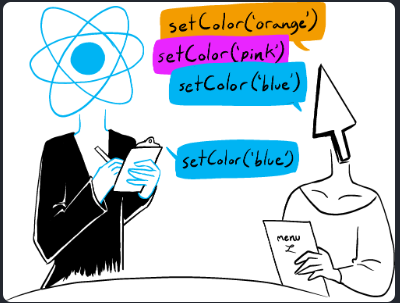
1. [Amazing summary of dataflow in react](https://dev.to/alexmercedcoder/react-data-flow-understanding-state-and-props-2759)Above link talks about 🡪 prop, state, state lifting, sharing data in: parent to child, child to parent, sibling, state drilling, deep drilling, solution of drilling 🡪 context, the cost of a context, and summary of all above and some more points. So give it a read, good for interview perspective.

Queueing a Series of State-Updates

Sometimes you might want to perform multiple operations on the value before queueing the next render. To do this, it helps to understand how React batches state updates.

As you might recall from the previous section, **each render’s state values are fixed**, so the value of number inside the first render’s event handler is always 0, no matter how many times you call setNumber(1):

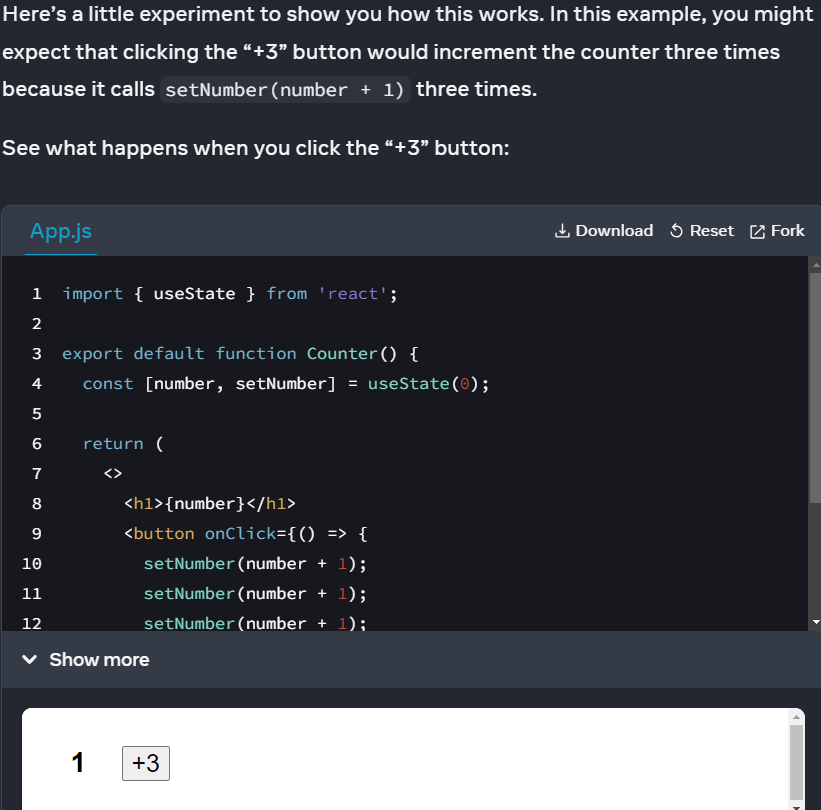
setNumber(0 + 1); setNumber(0 + 1); setNumber(0 + 1);

 **React waits until all code in the event handlers has run before processing your state updates. This is why the re-render *only* happens *after all these setNumber() calls.***

**A waiter doesn’t run to the kitchen at the mention of your first dish!** Instead, they let you finish your order, let you make changes to it, and even take orders from other people at the table.

1. **What “batching” is** and how React uses it to process multiple state updates
   * The UI won’t be updated until after your event handler, and any code in it, completes, known as batching.
   * It makes your React app run much faster. It also avoids dealing with confusing “half-finished” renders where only some of the variables have been updated.
   * React does not batch across multiple intentional events like clicks—each click is handled separately. Rest assured that React only does batching when it’s generally safe to do. This ensures that, for example, if the first button click disables a form, the second click would not submit it again.
2. [**How to apply several updates to the same state variable in a row**](https://youtu.be/bkRHbbYdOqs)If you would like to update the same state variable multiple times before the next render, instead of passing the next state value like setnumber(number + 1), you can pass a function that calculates the next state based on the previous one in the queue, like setnumber(n => n + 1).   
   It is a way to tell React to “do something with the state value” instead of just replacing it.

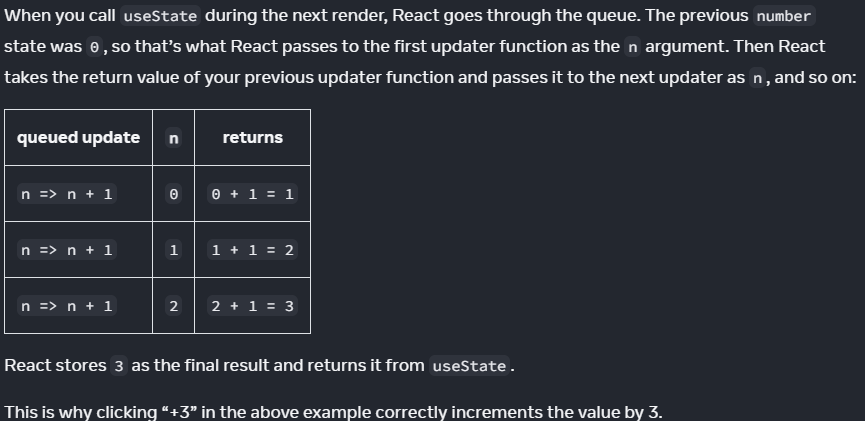
See the difference in output below when using arrow functions inside the setStateVariable thing(setNumber here) and without it.

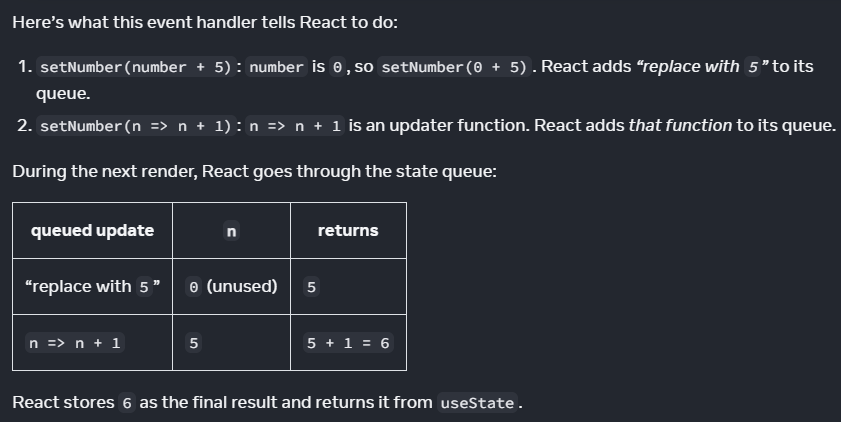
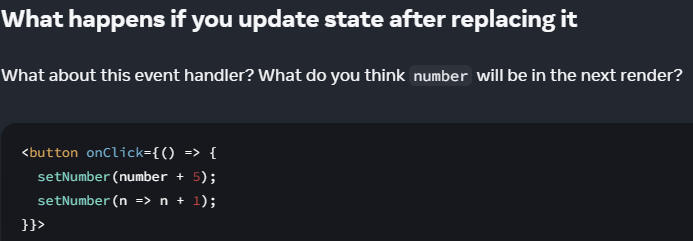
****

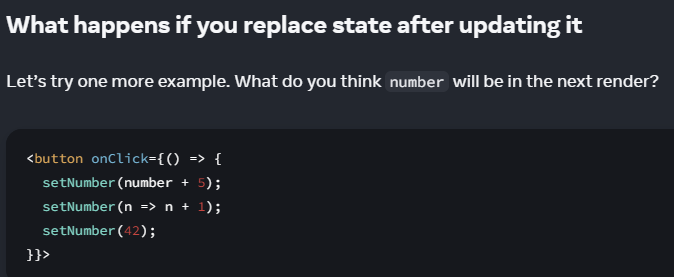
Here, n => n + 1 is called an updater function. When you pass it to a state setter:

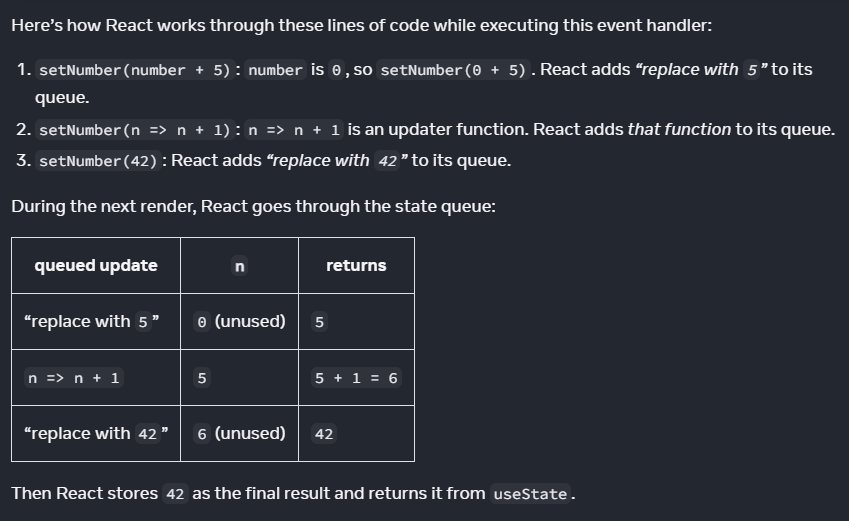
1. React queues this function to be processed after all the other code in the event handler has run.
2. During the next render, React goes through the queue and gives you the final updated state.

**Mechanism of how an updater function(arrow function inside setStateVariable function) works:**

****

****Trick question 1 :

Trick question 2 :



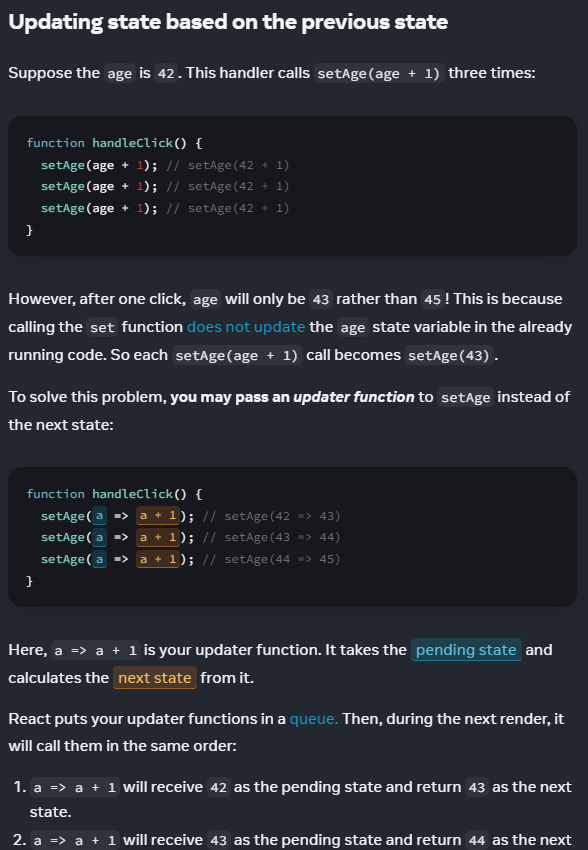
**To summarize,**

here’s how you can think of what you’re passing to the setNumber state setter:

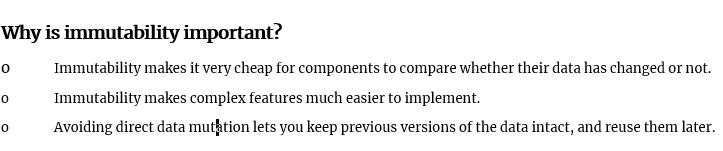
An updater function (e.g. n => n + 1) gets added to the queue. Any other value (e.g. number 5) adds “replace with 5” to the queue, ignoring what’s already queued. After the event handler completes, React will trigger a re-render. During the re-render, React will process the queue.

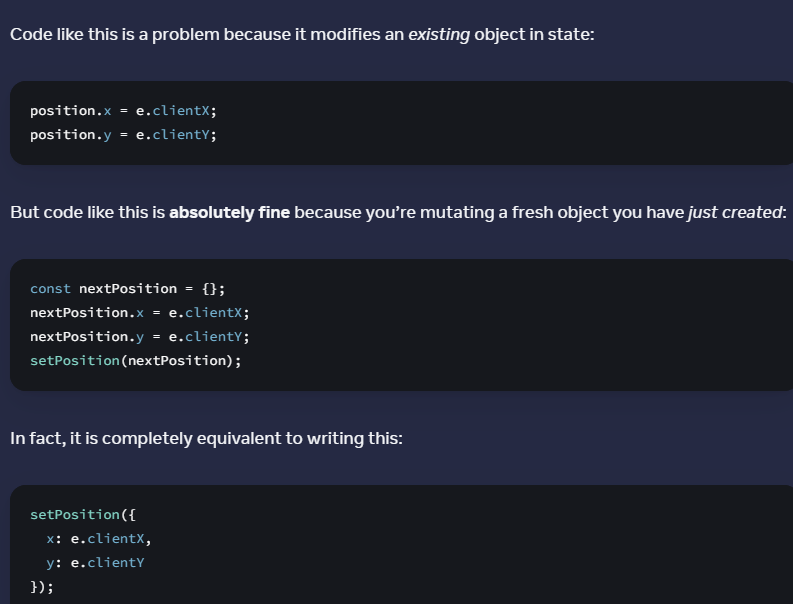
Updater functions run during rendering, so updater functions must be pure and only return the result. Don’t try to set state from inside of them or run other side effects.

**In Strict Mode, React will run each updater function twice (but discard the second result) to help you find mistakes.**

* Setting state does not change the variable in the existing render, but it requests a new render.
* React processes state updates after event handlers have finished running. This is called batching.
* To update some state multiple times in one event, you can use setNumber(n => n + 1) updater function.
* 

Updating Objects in State

**When you want to update an object, you need to create a new one (or make a copy of an existing one), and then set the state to use that copy.**

* You can store any kind of JavaScript value in state.  
    
  const [x, setX] = useState(0); setX(5);  
    
   The x state changed from 0 to 5, but the number 0 itself did not change. You trigger a re-render to replace a value. It’s not possible to make any changes to the built-in primitive values like numbers, strings, and Booleans in JavaScript.
* However, although objects in React state are technically mutable, you should treat them as if they were immutable—like numbers, booleans, and strings. Instead of mutating them, you should always replace them.
* 
* You can use the ... [object spread](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax#spread_in_object_literals) syntax so that you don’t need to copy every property separately. Note that the ... spread syntax is “shallow”—it only copies things one level deep. This makes it fast, but it also means that if you want to update a nested property, you’ll have to use it more than once.
* 
* If your state is deeply nested, you might want to consider [flattening](https://react.dev/learn/choosing-the-state-structure#avoid-deeply-nested-state) it. But, if you don’t want to change your state structure, you might prefer a shortcut to nested spreads. [Immer](https://github.com/immerjs/use-immer) is a popular library that lets you write using the convenient but mutating syntax and takes care of producing the copies for you. With Immer, the code you write looks like you are “breaking the rules” and mutating an object:
* updatePerson(draft => { draft.artwork.city = 'Lagos'; });
* Treat all state in React as immutable.
* When you store objects in state, mutating them will not trigger renders and will change the state in previous render “snapshots”.
* Instead of mutating an object, create a new version of it, and trigger a re-render by setting state to it.
* You can use the {...obj, something: 'newValue'} object spread syntax to create copies of objects.
* Spread syntax is shallow: it only copies one level deep.
* To update a nested object, you need to create copies all the way up from the place you’re updating.
* To reduce repetitive copying code, use Immer.

[ARRAY UPDATING METHODS](https://react.dev/learn/updating-arrays-in-state)

|  | **avoid (mutates the array)** | **prefer (returns a new array)** |
| --- | --- | --- |
| adding | push, unshift | concat, [...arr] spread syntax ([example](https://react.dev/learn/updating-arrays-in-state#adding-to-an-array)) |
| removing | pop, shift, splice | filter, slice ([example](https://react.dev/learn/updating-arrays-in-state#removing-from-an-array)) |
| replacing | splice, arr[i] = ... assignment | map ([example](https://react.dev/learn/updating-arrays-in-state#replacing-items-in-an-array)) |
| sorting | reverse, sort | copy the array first ([example](https://react.dev/learn/updating-arrays-in-state#making-other-changes-to-an-array)) |