Preserving and Resetting State

State is isolated between components. React keeps track of which state belongs to which component based on their place in the UI tree. You can control when to preserve state and when to reset it between re-renders.

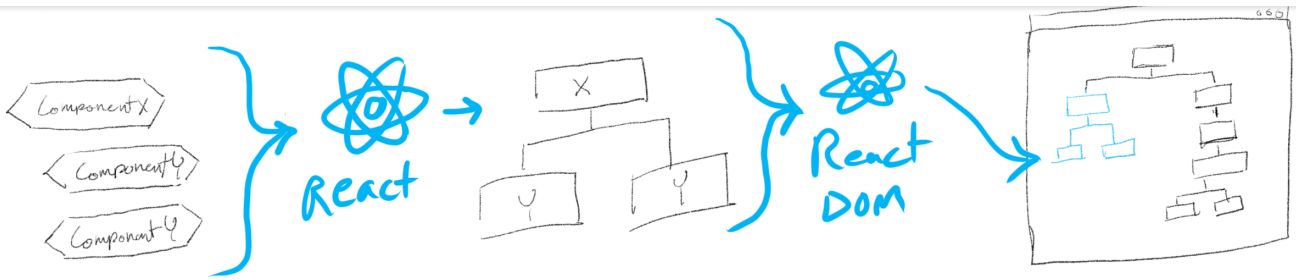
You will learn

- How React “sees” component structures
- When React chooses to preserve or reset the state
- How to force React to reset component’s state
- How keys and types affect whether the state is preserved

The UI tree

Browsers use many tree structures to model UI. The DOM represents HTML elements, the CSSOM does the same for CSS. There’s even an Accessibility tree!

React also uses tree structures to manage and model the UI you make. React makes **UI trees** from your JSX. Then React DOM updates the browser DOM elements to match that UI tree. (React Native translates these trees into elements specific to mobile platforms.)



State is tied to a position in the tree

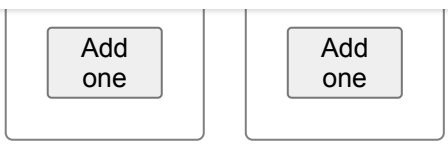
When you give a component state, you might think the state “lives” inside the component. But the state is actually held inside React. React associates each piece of state it’s holding with the correct component by where that component sits in the UI tree.

Here, there is only one `<Counter />` JSX tag, but it’s rendered at two different positions:

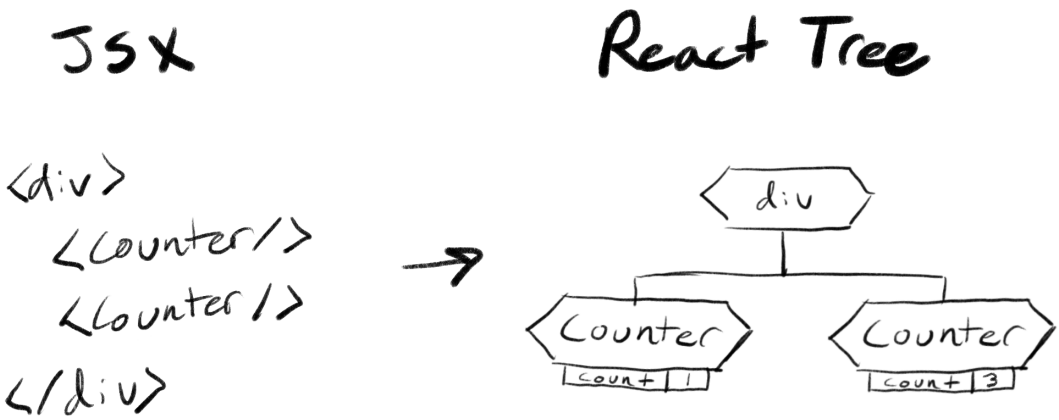
App.js Download Reset Fork

```
import { useState } from 'react';

export default function App() {
  const counter = <Counter />;
  return (
    <div>
      {counter}
      {counter}
    </div>
  );
}

function Counter() {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);
```

Show more



Here's how these look as a tree:



These are two separate counters because each is rendered at its own position in the tree. You don't usually have to think about these positions to use React, but it can be useful to understand how it works.

In React, each component on the screen has fully isolated state. For example, if you render two `Counter` components side by side, each of them will get its own, independent, `score` and `hover` states.

Try clicking both counters and notice they don't affect each other:

App.js Download Reset Fork

```
1  import { useState } from 'react';
2
3
4  export default function App() {
5    return (
6      <div>
7        <Counter />
8        <Counter />
9      </div>
10
11
12
13
```

```
function Counter() {
  const [score, setScore] = useState(0);
  const [hover, setHover] = useState(false);

  let className = 'counter';
```

Show more

React will only keep the state around for as long as you render the same component at the same position. To see this, increment both counters, then clear “Render the second counter” checkbox, and then tick it again:

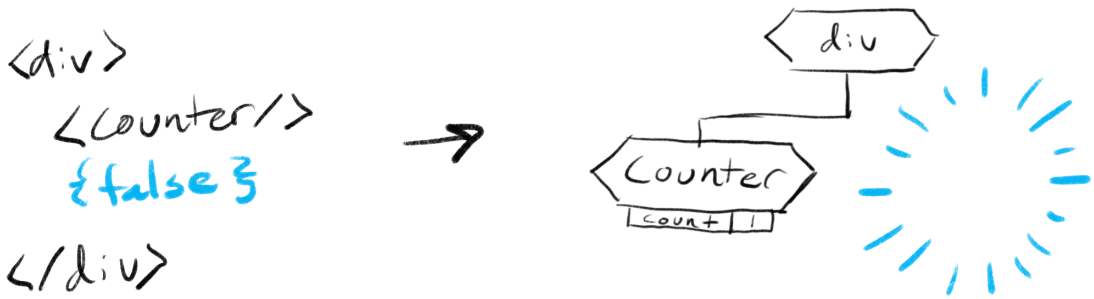
App.js

Download Reset Fork

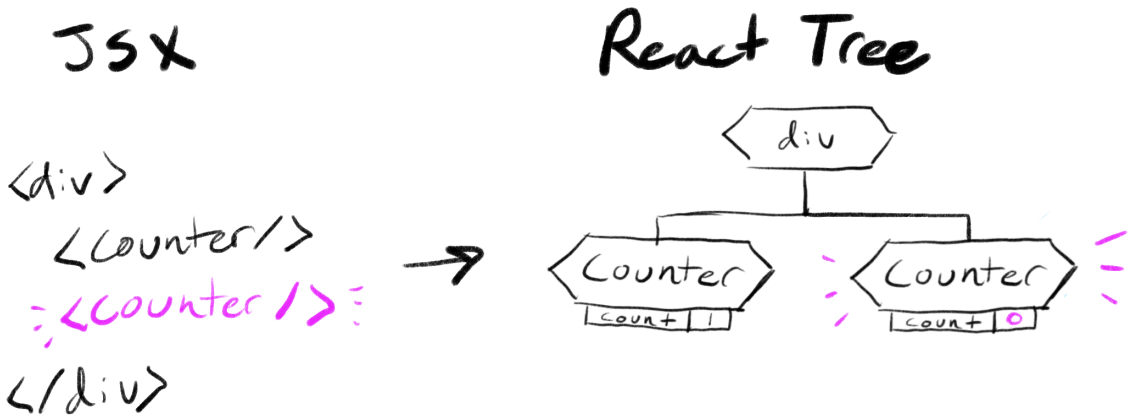
```
import { useState } from 'react';

export default function App() {
  const [showB, setShowB] = useState(true);
  return (
    <div>
      <Counter />
      {showB && <Counter />}
      <label>
        <input
          type="checkbox"
          checked={showB}
          onChange={e => {
            setShowB(e.target.checked)
          }}
        />
```

Show more

Notice how the moment you stop rendering the second counter, its state disappears completely. That’s because when React removes a component, it destroys its state.



When you tick “Render the second counter,” a second `Counter` and its state are initialized from scratch (`score = 0`) and added to the DOM.



React preserves a component’s state for as long as it’s being rendered at its position in the UI tree. If it gets removed, or a different component gets rendered at the same position, React discards its state.

Same component at the same position preserves state

In this example, there are two different `<Counter />` tags:

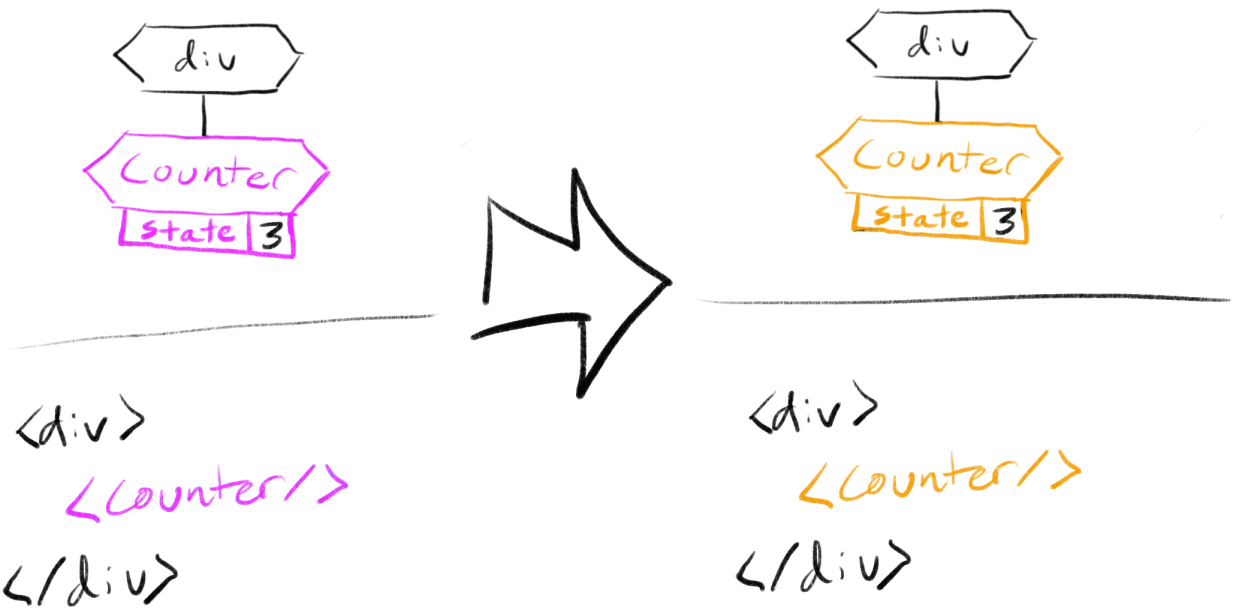
App.js Download Reset Fork

```
1  import { useState } from 'react';
2
```

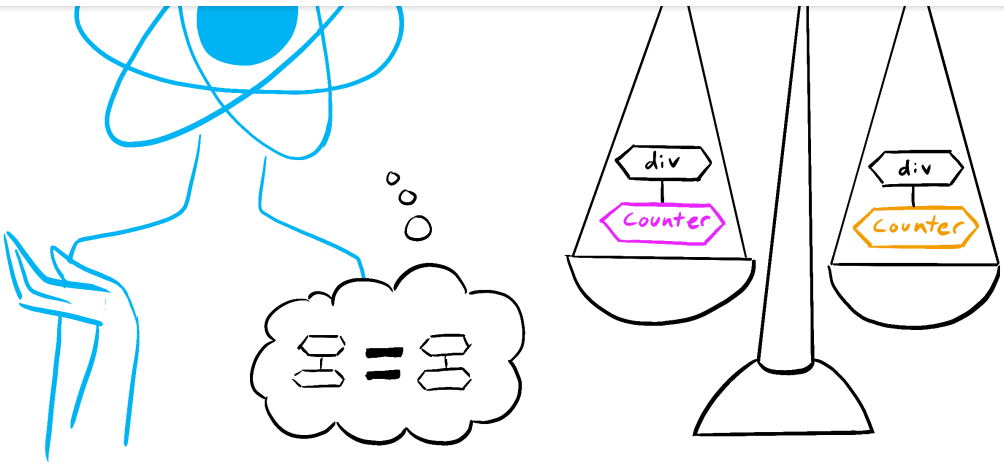
```
  return (
    <div>
      {isFancy ? (
        <Counter isFancy={true} />
      ) : (
        <Counter isFancy={false} />
      )}
      <label>
        <input
          type="checkbox"
          checked={isFancy}
```

Show more

When you tick or clear the checkbox, the counter state does not get reset. Whether `isFancy` is `true` or `false`, you always have a `<Counter />` as the first child of the `div` returned from the root `App` component:



It's the same component at the same position, so from React's perspective, it's the same counter.



Gotcha

Remember that **it's the position in the UI tree—not in the JSX markup—that matters to React!** This component has two `return` clauses with different `<Counter />` JSX tags inside and outside the `if`:

App.js Download Reset Fork

```
import { useState } from 'react';

export default function App() {
  const [isFancy, setIsFancy] = useState(false);
  if (isFancy) {
    return (
      <div>
        <Counter isFancy={true} />
        <label>
          <input
            type="checkbox"
            checked={isFancy}
            onChange={e => {
              setIsFancy(e.target.checked)
            }}
          />
```

You might expect the state to reset when you tick checkbox, but it doesn't! This is because **both of these `<Counter />` tags are rendered at the same position.** React doesn't know where you place the conditions in your function. All it "sees" is the tree you return. In both cases, the `App` component returns a `<div>` with `<Counter />` as a first child. This is why React considers them as *the same* `<Counter />`.

You can think of them as having the same "address": the first child of the first child of the root. This is how React matches them up between the previous and next renders, regardless of how you structure your logic.

Different components at the same position reset state

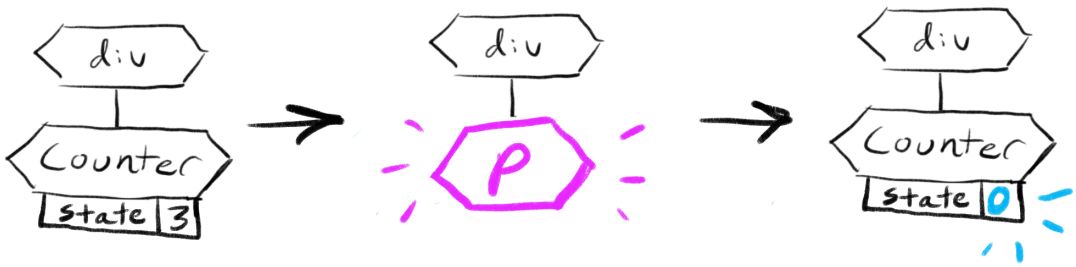
In this example, ticking the checkbox will replace `<Counter>` with a `<p>`:

App.js Download Reset Fork

```
import { useState } from 'react';

export default function App() {
  const [isPaused, setIsPaused] = useState(false);
  return (
    <div>
      {isPaused ? (
        <p>See you later!</p>
      ) : (
        <Counter />
      )}
      <label>
        <input
          type="checkbox"
          checked={isPaused}
          onChange={e => {
```

Show more

React removed the `Counter` from the UI tree and destroyed its state.



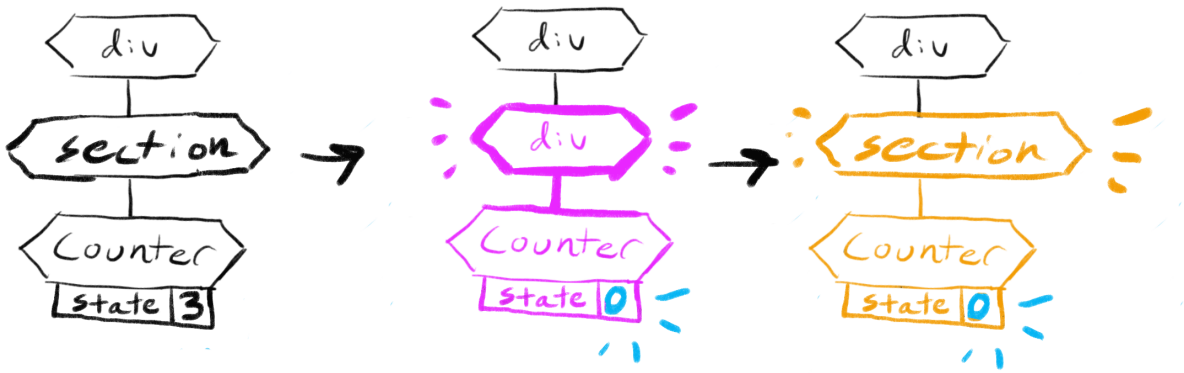
Also, **when you render a different component in the same position, it resets the state of its entire subtree.** To see how this works, increment the counter and then tick the checkbox:

App.js Download Reset Fork

```
import { useState } from 'react';

export default function App() {
  const [isFancy, setIsFancy] = useState(false);
  return (
    <div>
      {isFancy ? (
        <div>
          <Counter isFancy={true} />
        </div>
      ) : (
        <section>
          <Counter isFancy={false} />
        </section>
      )}
      <label>
```

Show more

The counter state gets reset when you click the checkbox. Although you render a `Counter`, the first child of the `div` changes from a `div` to a `section`. When the child



As a rule of thumb, **if you want to preserve the state between re-renders, the structure of your tree needs to "match up"** from one render to another. If the structure is different, the state gets destroyed because React destroys state when it removes a component from the tree.

Gotcha

This is why you should not nest component function definitions.

Here, the `MyTextField` component function is defined *inside* `MyComponent`:

App.js Download Reset Fork

```
import { useState } from 'react';

export default function MyComponent() {
  const [counter, setCounter] = useState(0);

  function MyTextField() {
    const [text, setText] = useState('');

    return (
      <input
        value={text}
        onChange={e => setText(e.target.value)}
```

Show more

Every time you click the button, the input state disappears! This is because a *different* `MyTextField` function is created for every render of `MyComponent`. You're rendering a *different* component in the same position, so React resets all state below. This leads to bugs and performance problems. To avoid this problem, **always declare component functions at the top level, and don't nest their definitions.**

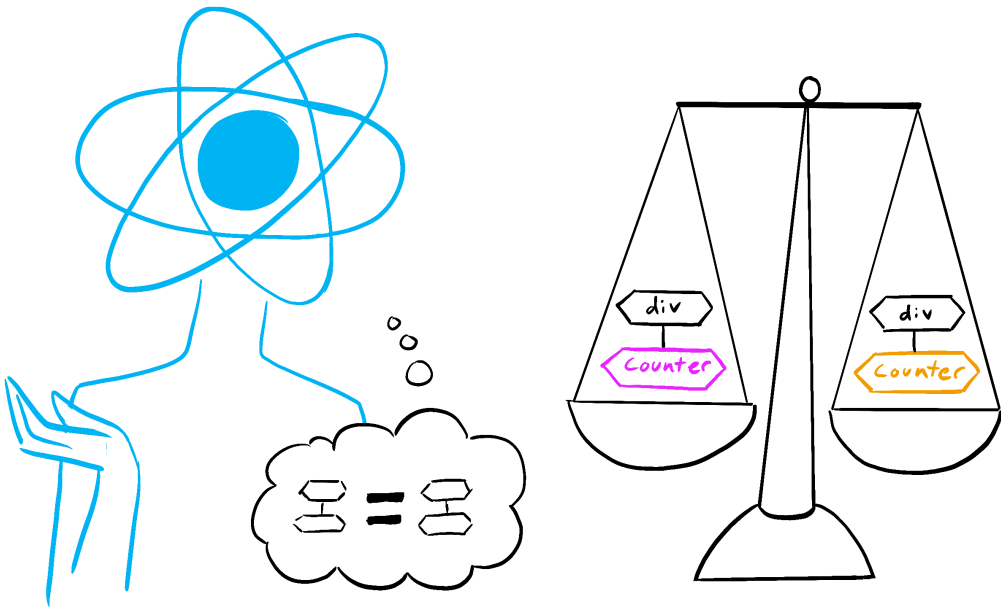
Resetting state at the same position

By default, React preserves state of a component while it stays at the same position. Usually, this is exactly what you want, so it makes sense as the default behavior. But sometimes, you may want to reset a component's state. Consider this app that lets two players keep track of their scores during each turn:

App.js Download Reset Fork

```
import { useState } from 'react';

export default function Scoreboard() {
  const [isPlayerA, setIsPlayerA] = useState(true);
  return (
    <div>
      {isPlayerA ? (
        <Counter person="Taylor" />
      ) : (
        <Counter person="Sarah" />
      )}
      <button onClick={() => {
        setIsPlayerA(!isPlayerA);
      }}>
        Next player!
      </button>
```

Currently, when you change the player, the score is preserved. The two `Counter`s appear in the same position, so React sees them as *the same* `Counter` whose `person` prop has changed.



But conceptually, in this app they should be two separate counters. They might appear in the same place in the UI, but one is a counter for Taylor, and another is a counter for Sarah.

There are two ways to reset state when switching between them:

1. Render components in different positions
2. Give each component an explicit identity with `key`

Option 1: Rendering a component in different positions

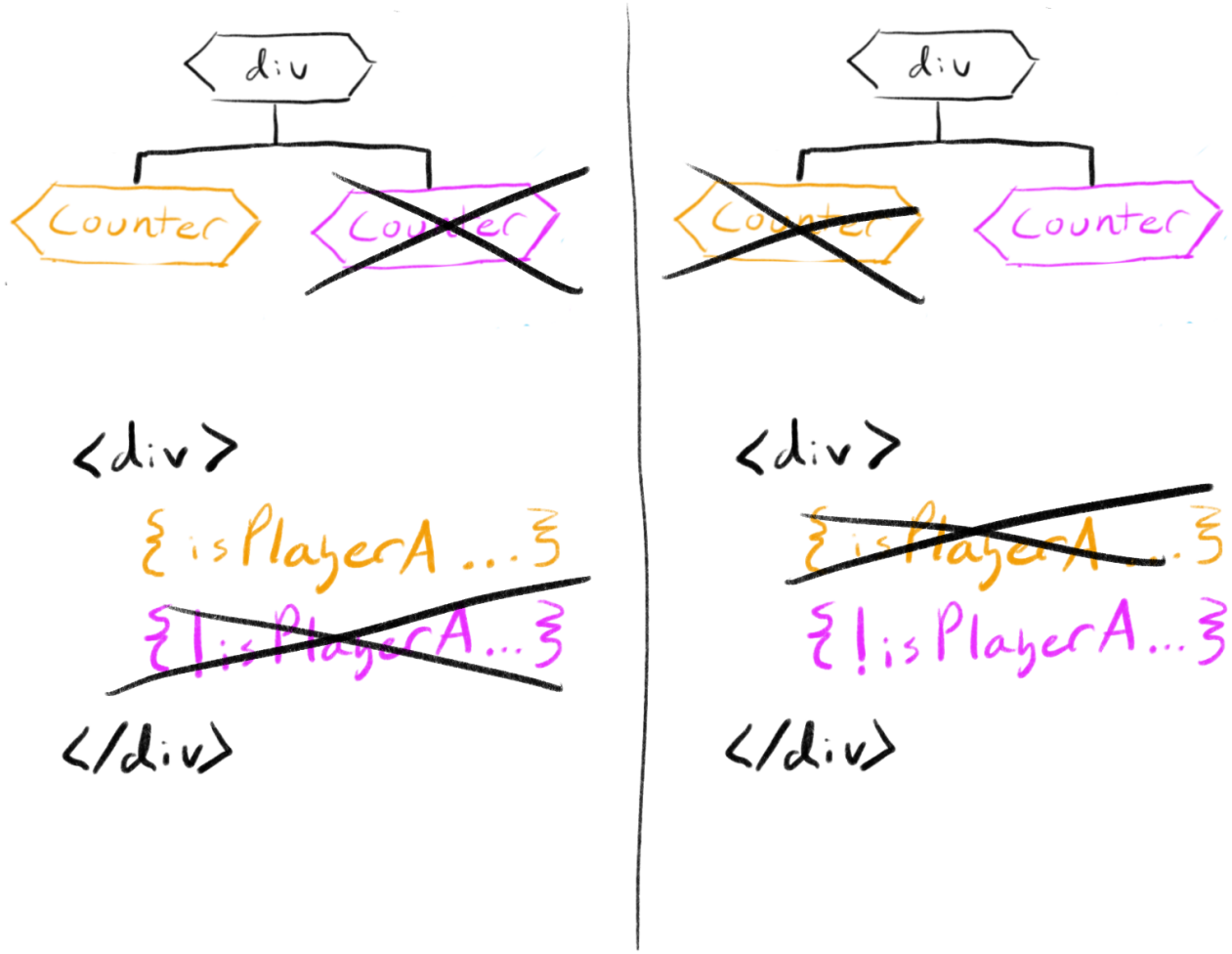
If you want these two `Counter`s to be independent, you can render them in two different positions:

App.js Download Reset Fork

```
  const [isPlayerA, setIsPlayerA] = useState(true);
  return (
    <div>
      {isPlayerA &&
        <Counter person="Taylor" />
      }
      {!isPlayerA &&
        <Counter person="Sarah" />
      }
      <button onClick={() => {
        setIsPlayerA(!isPlayerA);
      }}>
        Next player!
```

Show more

- Initially, `isPlayerA` is `true`. So the first position contains `Counter` state, and the second one is empty.
- When you click the “Next player” button the first position clears but the second one now contains a `Counter`.



Each `Counter`'s state gets destroyed each time its removed from the DOM. This is why they reset every time you click the button.

This solution is convenient when you only have a few independent components rendered in the same place. In this example, you only have two, so it's not a hassle to render both separately in the JSX.

Option 2: Resetting state with a key

There is also another, more generic, way to reset a component's state.

order within the parent ("first counter", "second counter") to discern between components. But keys let you tell React that this is not just a *first* counter, or a *second* counter, but a specific counter—for example, *Taylor's* counter. This way, React will know *Taylor's* counter wherever it appears in the tree!

In this example, the two `<Counter />` s don't share state even though they appear in the same place in JSX:

App.js Download Reset Fork

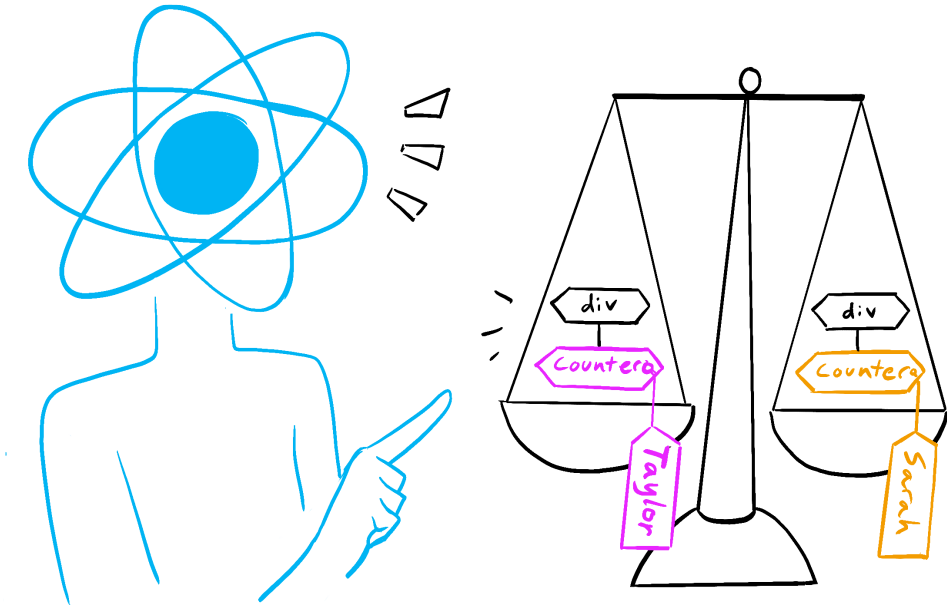
```
import { useState } from 'react';

export default function Scoreboard() {
  const [isPlayerA, setIsPlayerA] = useState(true);
  return (
    <div>
      {isPlayerA ? (
        <Counter key="Taylor" person="Taylor" />
      ) : (
        <Counter key="Sarah" person="Sarah" />
      )}
      <button onClick={() => {
        setIsPlayerA(!isPlayerA);
      }}>
        Next player!
      </button>
```

Show more

Switching between Taylor and Sarah does not preserve the state. This is because **you gave them different `key` s:**

```
{isPlayerA ? (
  <Counter key="Taylor" person="Taylor" />
) : (
  <Counter key="Sarah" person="Sarah" />
```

Specifying a `key` tells React to use the `key` itself as part of the position, instead of their order within the parent. This is why, even though you render them in the same place in JSX, from React's perspective, these are two different counters. As a result, they will never share state. Every time a counter appears on the screen, its state is created. Every time it is removed, its state is destroyed. Toggling between them resets their state over and over.



Remember that keys are not globally unique. They only specify the position *within the parent*.

Resetting a form with a key

Resetting state with a key is particularly useful when dealing with forms.

In this chat app, the `<Chat>` component contains the text input state:

App.js ContactList.js Chat.js Reset Fork

```
import ContactList from './ContactList.js';

export default function Messenger() {
  const [to, setTo] = useState(contacts[0]);
  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedContact={to}
        onSelect={contact => setTo(contact)}
      />
      <Chat contact={to} />
    </div>
  )
```

Show more

Try entering something into the input, and then press “Alice” or “Bob” to choose a different recipient. You will notice that the input state is preserved because the `<Chat>` is rendered at the same position in the tree.

In many apps, this may be the desired behavior, but not in a chat app! You don’t want to let the user send a message they already typed to a wrong person due to an accidental click. To fix it, add a `key`:

```
<Chat key={to.id} contact={to} />
```

This ensures that when you select a different recipient, the `Chat` component will be recreated from scratch, including any state in the tree below it. React will also re-create the DOM elements instead of reusing them.

Now switching the recipient always clears the text field:

App.js ContactList.js Chat.js Reset Fork

```
export default function Messenger() {
  const [to, setTo] = useState(contacts[0]);
  return (
    <div>
      <ContactList
        contacts={contacts}
        selectedContact={to}
        onSelect={contact => setTo(contact)}
      />
      <Chat key={to.id} contact={to} />
    </div>
  )
```

Show more

DEEP DIVE

Preserving state for removed components

Show Details

Recap

- React keeps state for as long as the same component is rendered at the same position.
- State is not kept in JSX tags. It's associated with the tree position in which you put that JSX.
- You can force a subtree to reset its state by giving it a different key.
- Don't nest component definitions, or you'll reset state by accident.

React Docs

Try out some challenges

1. Fix disappearing input text | 2. Swap two form fields | 3. Reset a detail form | 4. Cle

Challenge 1 of 5:
Fix disappearing input text

This example shows a message when you press the button. However, pressing the button also accidentally resets the input. Why does this happen? Fix it so that pressing the button does not reset the input text.

App.js

Reset Fork

```
import { useState } from 'react';

export default function App() {
  const [showHint, setShowHint] = useState(false);
  if (showHint) {
    return (
      <div>
        <p><i>Hint: Your favorite city?</i></p>
        <Form />
        <button onClick={() => {
          setShowHint(false);
        }}>Hide hint</button>
      </div>
    );
  }
  return (
```

Show more

Show solution

Next Challenge

PREVIOUS

NEXT
Extracting State Logic into a Reducer

FACEBOOK

Open Source