Undertanging setState in depth

React components with state render based on that state. When the state of components changes, so does the component .
That makes understanding when and how to change the state of your component important. At the end of this tutorial, you should know how setState works, and be able to avoid common pitfalls that many of us hit when when learning React.

Workings of `setState()`

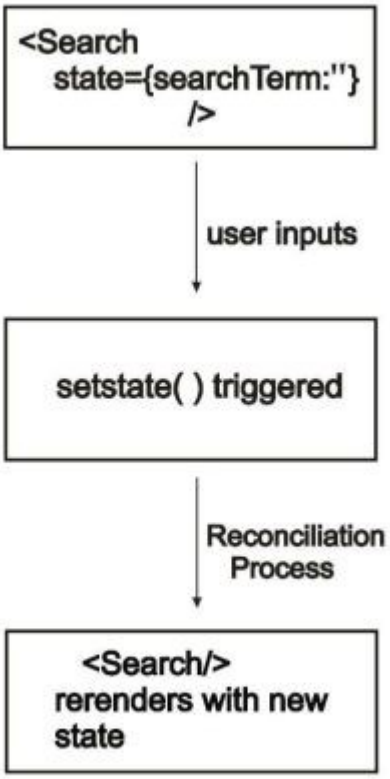
setState() is the only legitimate way to update state after the initial state setup. Let's say we have a search component and want to display the search term a user submits.
Here's the setup:

```
import React, { Component } from 'react'


class Search extends Component {
  constructor(props) {
    super(props)

    state = {
      searchTerm: ''
    }
  }
}
```

We're passing an empty string as a value and, to update the state of searchTerm, we have to call setState().

```
setState({ searchTerm: event.target.value })
```

Here, we're passing an object to setState(). The object contains the part of the state we want to update which, in this case, is the value of searchTerm. React takes this value and merges it into the object that needs it. It's sort of like the Search component asks what it should use for the value of searchTerm and setState() responds with an answer.



This is basically kicking off a process that React calls reconciliation. The reconciliation process is the way React updates the DOM, by making changes to the component based on the change in state. When the request to setState() is triggered, React creates a new tree containing the reactive elements in the component (along with the updated state). This tree is used to figure out how the Search component's should change in response to the state change by comparing it with the elements of the previous tree. React knows which changes to implement and will only update the parts of the DOM where necessary. This is why React is fast.
That sounds like a lot, but to sum up the flow:

•We have a search component that displays a search term

•That search term is currently empty

•The user submits a search term

•That term is captured and stored by setState as a value

•Reconciliation takes place and React notices the change in value

•React instructs the search component to update the value and the search term is merged in

The reconciliation process does not necessarily change the entire tree, except in a situation where the root of the tree is changed like this:

```
// old
<div>
  <Search />
</div>

// new
<span>
  <Search />
</span>
```

All tags become tags and the whole component tree will be updated as a result.
The rule of thumb is to never mutate state directly. Always use setState() to change state. Modifying state directly, like the snippet below will not cause the component to re-render.

```
// do not do this
this.state = {
  searchTerm: event.target.value
}
```

Passing a Function to `setState()`

To demonstrate this idea further, let’s create a simple counter that increments and decrements on click.

Let’s register the component and define the markup for the :

```
class App extends React.Component {

state = { count: 0 }

handleIncrement = () => {
  this.setState({ count: this.state.count + 1 })
}

handleDecrement = () => {
  this.setState({ count: this.state.count - 1 })
}
  render() {
    return (
      <div>
        <div>
          {this.state.count}
        </div>
        <button onClick={this.handleIncrement}>Increment by 1</button>
        <button onClick={this.handleDecrement}>Decrement by 1</button>
      </div>
    )
  }
}
```

At this point, the counter simply increments or decrements the count by 1 on each click.
But what if we wanted to increment or decrement by 3 instead?
We could try to call setState() three times in the handleDecrement and handleIncrement functions like this:

```
handleIncrement = () => {
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
  this.setState({ count: this.state.count + 1 })
}

handleDecrement = () => {
  this.setState({ count: this.state.count - 1 })
  this.setState({ count: this.state.count - 1 })
  this.setState({ count: this.state.count - 1 })
}
```

If you are coding along at home, you might be surprised to find **that doesn't work.**
The above code snippet is equivalent to:

```
Object.assign(
  {},
  { count: this.state.count + 1 },
  { count: this.state.count + 1 },
  { count: this.state.count + 1 },
)
```

Object.assign() is used to copy data from a source object to a target object. If the data being copied from the source to the target all have same keys, like in our example, the last object wins. Here's a simpler version of how Object.assign() works;

```
let count = 3

const object = Object.assign({},
  {count: count + 1},
  {count: count + 2},
  {count: count + 3}
);

console.log(object);
// output: Object { count: 6 }
```

So instead of the call happening three times, it happens just once. This can be fixed by passing a function to setState(). Just as you pass objects to setState(), you can also pass functions, and that is the way out of the situation above.

If we edit the handleIncrement function to look like this:

```
handleIncrement = () => {
  this.setState((prevState) => ({ count: prevState.count + 1 }))
  this.setState((prevState) => ({ count: prevState.count + 1 }))
  this.setState((prevState) => ({ count: prevState.count + 1 }))
}
```

...we can now increment count three times with one click.
In this case, instead of merging, React queues the function calls in the order they are made and updates the entire state ones it is done. This updates the state of count to 3 instead of 1.

Access Previous State Using Updater

When building React applications, there are times when you'll want to calculate state based the component's previous state. You cannot always trust this.state to hold the correct state immediately after calling setState(), as it is always equal to the state rendered on the screen.

Let's go back to our counter example to see how this works. Let's say we have a function that decrements our count by 1. This function looks like this:

```
changeCount = () => {
  this.setState({ count: this.state.count - 1})
}
```

What we want is the ability to decrement by 3. The changeCount() function is called three times in a function that handles the click event, like this.

```
handleDecrement = () => {
  this.changeCount()
  this.changeCount()
  this.changeCount()
}
```

Each time the button to decrement is clicked, the count will decrement by 1 instead of 3. This is because the this.state.count does not get updated until the component has been re-rendered. The solution is to use an updater. An updater allows you access the current state and put it to use immediately to update other items. So the changeCount() function will look like this.

```
changeCount = () => {
  this.setState((prevState) => {
    return { count: prevState.count - 1}
  })
}
```

Now we are not depending on the result of this.state. The states of count are built on each other so we are able to access the correct state which changes with each call to changeCount(). setState() should be treated asynchronously — in other words, do not always expect that the state has changed after calling setState().

Wrapping Up

When working with setState(), these are the major things you should know:

•Update to a component state should be done using setState()

•You can pass an object or a function to setState()

•Pass a function when you can to update state multiple times

•Do not depend on this.state immediately after calling setState()

and make use of the updater function instead.