# React State and Arrays

(https://github.com/learn-co-curriculum/react-hooks-state-arrays) (https://github.com/learn-co-curriculum/react-hooks-state-arrays/issues/new)

## Learning Goals

- Understand how to work with arrays in state
- Add elements to arrays in state
- Remove elements from arrays in state
- Update elements in arrays in state
- Conditionally render elements from arrays in state

## Introduction

In this lesson, we'll discuss how to work with arrays in state. Working with arrays and objects in state requires special care because of one simple rule:

**React will only update state if a new object/array is passed to setState.**

In this lesson, we'll learn some common patterns for creating new arrays when we need to add elements to an array in state, remove elements from arrays in state, and update individual items in arrays in state.

Fork and clone this lesson so you can code along!

## Understand Why We Can't Mutate State

From the React documentation on state:

> State can hold any kind of JavaScript value, including objects. But you shouldn't change objects that you hold in the React state directly. Instead, 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.
>
> — **[React Docs: Updating Objects in State](https://beta.reactjs.org/learn/updating-objects-in-state) (https://beta.reactjs.org/learn/updating-objects-in-state)**

What does this mean? Consider the following example of a stateful component:

```
function Counter() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount((count) => count + 1);
  }

  return <button onClick={handleClick}>{count}</button>;
}
```

In an earlier lesson, we learned that when the button is clicked in this example component, the following occurs:

1. Calling `setCount(1)` tells React that its internal state for our `Counter` component's `count` value must update to 1
2. React updates its internal state
3. React **re-renders** the `Counter` component based on changes to internal state
4. When the `Counter` component is re-rendered, `useState` will return the current value of React's internal state, which is now 1
5. The value of `count` is now 1 within our `Counter` component
6. Our component's JSX uses this new value to display the number 1 within the button

In step 3 above, one key detail is that React re-renders components based on *changes* to state. For example, if we updated the code of our component in such a way that instead of incrementing the count when the button was clicked, we simply called `setCount` *with the same value*, React wouldn't re-render our component, since *we gave it the same value for state* as it already has:

```
function Counter() {
  const [count, setCount] = useState(0);

  function handleClick() {
    // setting state with the same value
    setCount(count);
  }

  return <button onClick={handleClick}>{count}</button>;
}
```

This detail is a small one, but crucial to understanding how state works in React! Consider the following refactor. Let's use an object instead of a number for our initial state:

```
function Counter() {
  const [count, setCount] = useState({ x: 0 });

  function handleClick() {
    // increment the count
    count.x++;
    // set state with the new count value
    setCount(count);
  }

  return <button onClick={handleClick}>{count.x}</button>;
}
```

If you try running this example code, you'll see that the component doesn't actually rerender with a new value. Why is this? Well, we're not actually providing a new value when we're calling `setCount` ! Just like in the previous example, setting state with the same value will not cause React to re-render our component. Even though we've *mutated* the value of the `count` object, the `count` object is still the same object in memory.

You may have seen examples of this behavior in JavaScript previously:

```
const count = { x: 0 }; // create an object and save a reference to it as "count"

const alsoCount = count; // save a new reference to the same "count" object to "alsoCount"
```

```
console.log(alsoCount === count); // true, both "count" and "alsoCount" point to the same object in memory

count.x++; // mutate the count object

console.log(count.x === alsoCount.x); // true, both "count" and "alsoCount" point to the same object in memory
```

So then, how can we work with objects in state in React? The answer is:

**Any time we need to update an object or array in state, we need to make a new object and call `setState` with the new object**.

For our counter example, here's how that would work:

```
function Counter() {
  const [count, setCount] = useState({ x: 0 });

  function handleClick() {
    // set state with a new object
    setCount({ x: count.x + 1 });
  }

  return <button onClick={handleClick}>{count.x}</button>;
}
```

In the example above, we are successfully updating state by creating a new object with the incremented value, and setting state with that new object.

We must also follow this rule when working with arrays, and find ways to update array values without mutating state. In the rest of this lesson, we'll show some common patterns for this.

## Adding Elements To Arrays In State

When we need to represent a list of data in our UI, it's often a good idea to have the data for that list stored in an array! To give an example, let's build out a component that does the following:

- Shows a button to generate a new spicy food
- When the button is clicked, adds the newly generated food to a list

The starter code for this component is in `SpicyFoodList.js` . Before we walk through the solution, see if you can get this working by **adding a new spicy food to the array when the button is clicked**.

Recall that when working with objects and arrays in state:

**React will only update state if a new object is passed to `setState` .**

That means it's important to keep in mind which array methods *mutate* arrays, and which can be used to make *copies* of arrays.

For this deliverable, the goal is to create a new copy of the array which includes all the elements of the original array, plus a new element.

Ok, give it a shot! Then scroll down to see a solution.

...

...

...

...

...

...

...

...

So, how can we update state and get new foods to display dynamically? Here's the code:

```
function handleAddFood() {
  const newFood = getNewRandomSpicyFood();
  const newFoodArray = [...foods, newFood];
```

```
    setFoods(newFoodArray);
  }
```

This step is crucial, so let's break it down:

```
const newFoodArray = [...foods, newFood];
```

Here, we're using the spread operator ( `...` ) to make a *copy* of our `foods` array, and insert each element into a *new* array. We're also adding the newly generated food returned by the `getNewRandomSpicyFood` function at the end of the array.

Remember, whenever we are updating state, it's important that we always pass a new object/array to `setState` . That's why we're using the spread operator here to make a copy of the array, instead of `.push` , which will mutate the original array.

Again, to repeat! React will *only* re-render our component when we set state with a *new* value; so we need to create a new **copy** of our original array to pass to the setter function, rather than mutating the original array directly and passing a reference to the original array.

After setting state, our component should automatically re-render with the new list of foods.

## Removing Elements From Arrays In State

Let's add another feature. When a user clicks on a food, that food should be *removed* from the list.

First, we'll need to add a click handler to the `<li>` elements, and pass in the id of the food we're trying to remove:

```
const foodList = foods.map((food) => (
  <li key={food.id} onClick={() => handleLiClick(food.id)}>
    {food.name} | Heat: {food.heatLevel} | Cuisine: {food.cuisine}
  </li>
));
```

Next, in the `handleLiClick` function, we need to figure out a way to update our array in state so it no longer includes the food.

There are a few approaches you could take here, so try to find a solution on your own before peeking at the answer! Remember, we want to find a way to remove the food by *creating a new array that has all the original elements, except the one we want removed*.

...

...

...

...

...

...

...

...

One common approach to this problem of creating a new array that doesn't include a specific element is using the `.filter` method. Here's how we can do it:

```
function handleLiClick(id) {
  const newFoodArray = foods.filter((food) => food.id !== id);
  setFoods(newFoodArray);
}
```

Calling `.filter` will return a *new array* based on which elements match our criteria in the callback function. So if we write our callback function in `.filter` to look for all foods *except* the number we're trying to remove, we'll get back a new, shortened list of foods:

```
[1, 2, 3].filter((number) => number !== 3);
// => [1,2]
```

Setting state with this updated list of foods will re-render our component, causing the food to be removed from the list.

## Updating Elements in Arrays in State

Here's a tough one! We've seen how to add and remove elements from arrays, but what about updating them?

Let's update our click feature so that when a user clicks on a food, that food's heat level is incremented by 1.

In the `handleLiClick` function, we need to figure out a way to update our array in state and increment the heat level *only* for the food that was clicked.

Once again, there are a few approaches you could take here, so try to find a solution on your own before peeking at the answer! Remember, we want to find a way to update the heat level by *creating a new array that has all the elements from the original array, with one element updated*.

...

...

...

...

...

...

...

...

One approach we can take to *updating* items in arrays by creating a new array involves using the `.map` method. Calling `.map` will return a new array with the same length as our original array (which is what we want), with some transformations applied to the elements in the array.

Here's an example of using `.map` to update *one element* of an array:

```
[1, 2, 3].map((number) => {
  if (number === 3) {
    // if the number is the one we're looking for, increment it
    return number + 100;
  } else {
    // otherwise, return the original number
    return number;
  }
});
// => [1,2,103]
```

So to use that technique to solve our problem, here's how our click event handler would look:

```
function handleLiClick(id) {
  const newFoodArray = foods.map((food) => {
    if (food.id === id) {
      return {
        ...food,
        heatLevel: food.heatLevel + 1,
      };
    } else {
      return food;
    }
  });
  setFoods(newFoodArray);
}
```

To break down the steps:

- `.map` will iterate through the array and return a new array
- Whatever value is returned by the callback function that we pass to `.map` will be added to this new array
- If the ID of the food we're iterating over matches the ID of the food we're updating, return a new food object with the heat level incremented by 1
- Otherwise, return the original food object

## Array Cheat Sheet

Here's a quick reference of some common techniques for manipulating arrays in state. Keep this in mind, because working with arrays will be important as a React developer!

- **Add**: use the spread operator ( `[...]` )
- **Remove**: use `.filter`
- **Update**: use `.map`

## Working With Multiple State Variables

Sometimes, a component needs multiple state variables to represent multiple UI states. To give an example, let's add a feature to our `SpicyFoodList` component that lets the user filter the list to only show foods from a certain cuisine.

Here's the JSX you'll need for this feature:

```
<select name="filter">
  <option value="All">All</option>
  <option value="American">American</option>
  <option value="Sichuan">Sichuan</option>
  <option value="Thai">Thai</option>
  <option value="Mexican">Mexican</option>
</select>
```

Try building out this feature on your own, then we'll go through the solution. Think about what new *state variable* you'll need to add, and how to use that variable to determine which foods are being displayed!

...

...

...

...

...

...

...

...

Let's start by talking through what new state we'll need to add. We need some way of keeping track of which option the user selected from the `<select>` tag. We'll also need to use that data to *filter* the list of foods to determine which ones to display.

Let's set up our initial state to be a string of "All" to match the first `<option>` in our dropdown:

```
const [filterBy, setFilterBy] = useState("All");
```

With this state variable in place, we can update the `<select>` element to set the `filterBy` variable when its value is changed, like so:

```
function handleFilterChange(event) {
  setFilterBy(event.target.value);
}

return (
  <select name="filter" onChange={handleFilterChange}>
    <option value="All">All</option>
    <option value="American">American</option>
    <option value="Sichuan">Sichuan</option>
    <option value="Thai">Thai</option>
    <option value="Mexican">Mexican</option>
  </select>
);
```

Next, let's figure out how this filter value can be used to update what foods are displayed. We will need to use *both* of our state variables together to solve this problem! Here's how we can use the filter value to update which items are displayed:

```
const [foods, setFoods] = useState(spicyFoods);
const [filterBy, setFilterBy] = useState("All");

const foodsToDisplay = foods.filter((food) => {
  if (filterBy === "All") {
    return true;
  } else {
    return food.cuisine === filterBy;
  }
});
```

> Remember, `.filter` returns a new array that is a shortened version of the elements in the original array. It expects a callback that will return either true or false. For all elements of the original array where the callback returns true, add those elements to the new array. For all elements that return false, don't add them to the new array.

This will give us a new variable, `foodsToDisplay` , that will be an array of:

- All foods from `foods` array if `filterBy` is set to "All"
- Only foods that match the cuisine in `filterBy` if `filterBy` is not set to "All"

Now, we just need to use `foodsToDisplay` instead of `foods` when we're generating the `<li>` elements:

```
const foodList = foodsToDisplay.map((food) => (
  <li key={food.id} onClick={() => handleLiClick(food.id)}>
    {food.name} | Heat: {food.heatLevel} | Cuisine: {food.cuisine}
  </li>
));
```

Having both of these variables in state and knowing how to use them in conjunction with each other gives us a lot of power in React! All we need to worry about is using our programming tools — working with *data*; manipulating *arrays* — and React can take care of all the hard work of updating the DOM correctly.

## Conclusion

When working with arrays in state, it's important to find ways to set state by passing in a new array rather than mutating the original array. That means using array methods like `map` and `filter` , or the spread operator, to create copies of arrays before setting state.

## Resources

- **React Docs (beta): Updating Arrays in State (https://beta.reactjs.org/learn/updating-arrays-in-state)**
- **React State Arrays (https://www.robinwieruch.de/react-state-array-add-update-remove)**