React Lecture - 12

Objects and Arrays in States

Objects in states

Introductions

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.

Updating a state object

Although object states are mutable but it is not recommended to mutate object state because directly mutation the state will not trigger a re render and the data will not update on our web page.
We should instead use the state setter function to update objects.

We always **pass a new object** to the state setter function.

```
const [position, setPosition] = useState({
  x: 0,
  y: 0
});
return (
```

```
setPosition({
  x: 5,
  y: 6
});
```

Updating or Adding properties in a state object

Often, you will want to include existing data as a part of the new object you're creating.

For example, you may want to update only one field in a form, but keep the previous values for all other fields.

To update only one property in the object you need to copy the object in a new object and then override the value of the property you want to change.

```
export default function Form() {
  const [person, setPerson] = useState({
    firstName: 'Barbara',
    lastName: 'Hepworth',
    email: 'bhepworth@sculpture.com'
  });
```

```
setPerson({
  ...person, // Copy the old fields
  firstName: e.target.value // But override this one
});
```

Updating or Adding properties to nested objects

Consider a nested object structure like this: Suppose we want to modify the value of person.artwork.title

In order to change city, you would first need to produce the new artwork object (pre-populated with data from the previous one), and then produce the new person object which points at the new artwork:

```
const [person, setPerson] = useState({
  name: 'Niki de Saint Phalle',
  artwork: {
    title: 'Blue Nana',
    city: 'Hamburg',
    image: 'https://i.imgur.com/Sd1AgUOm.jpg',
  }
});
```

Updating or Adding properties to nested objects

```
setPerson({
  ...person, // Copy other fields
  artwork: { // but replace the artwork
    ...person.artwork, // with the same one
    city: 'New Delhi' // but in New Delhi!
  }
});
```

Updating Arrays in State

Introduction

Arrays are mutable in JavaScript, but you should treat them as immutable when you store them in state. Just like with objects, when you want to update an array stored in state, you need to create a new one (or make a copy of an existing one), and then set state to use the new array.

Updating Arrays without mutation

Every time you want to update an array, you'll want to pass a new array to your state setter function. To do that, you can create a new array from the original array in your state by calling its non-mutating methods like filter() and map(). Then you can set your state to the resulting new array.

Updating Arrays without mutation

Here is a reference table of common array operations. When dealing with arrays inside React state, you will need to avoid the methods in the left column, and instead prefer the methods in the right column:

	avoid (mutates the array)	prefer (returns a new array)
adding	`push`, `unshift`	`concat`, `[...arr]` spread syntax (example)
removing	`pop`, `shift`, `splice`	`filter`, `slice` (example)
replacing	`splice`, `arr[i] = ...` assignment	`map` (example)
sorting	`reverse`, `sort`	copy the array first (example)

Alternatively, you can use Immer which lets you use methods from both columns.

Adding to array

Create a new array which contains the existing items and a new item at the end.

There are multiple ways to do this, but the easiest one is to use the ... array spread syntax:

```
setArtists( // Replace the state
  [ // with a new array
    ...artists, // that contains all the old items
    { id: nextId++, name: name } // and one new item at the end
  ]
);
```

The array spread syntax also lets you prepend an item by placing it before the original ...artists:

Removing from an array

The easiest way to remove an item from an array is to filter it out. In other words, you will produce a new array that will not contain that item.

```
setArtists(
  artists.filter(a => a.id !== artist.id)
);
```

Transforming an array

If you want to change some or all items of the array, you can use map() to create a new array. The function you will pass to map can decide what to do with each item, based on its data or its index (or both).

Replacing items in an array

It is particularly common to want to replace one or more items in an array. Assignments like arr[o] = 'bird' are mutating the original array, so instead you'll want to use map for this as well.

To replace an item, create a new array with map. Inside your map call, you will receive the item index as the second argument. Use it to decide whether to return the original item (the first argument) or something else:

Inserting into array

Sometimes, you may want to insert an item at a particular position that's neither at the beginning nor at the end. To do this, you can use the ... array spread syntax together with the slice() method.

The slice() method lets you cut a "slice" of the array.

To insert an item, you will create an array that spreads the slice before the insertion point, then the new item, and then the rest of the original array.

```
const insertAt = 1; // Could be any index
const nextArtists = [
  // Items before the insertion point:
  ...artists.slice(0, insertAt),
  // New item:
  { id: nextId++, name: name },
  // Items after the insertion poiSnt:
  ...artists.slice(insertAt)
];
setArtists(nextArtists);
```

Making other changes to an array

There are some things you can't do with the spread syntax and non-mutating methods like map() and filter() alone.

For example, you may want to reverse or sort an array. The JavaScript reverse() and sort() methods are mutating the original array, so you can't use them directly.

However, you can copy the array first, and then make changes to it.