W8D2

Most important thing to remember from yesterday is making nice, simple, testable components. You can then use these Lego blocks to build up to a big app.

Finite State Machine (Reducer functions)

Finite State Machine sounds complex, but really isn't! It's a computer science thing.

Think of a turnstile:

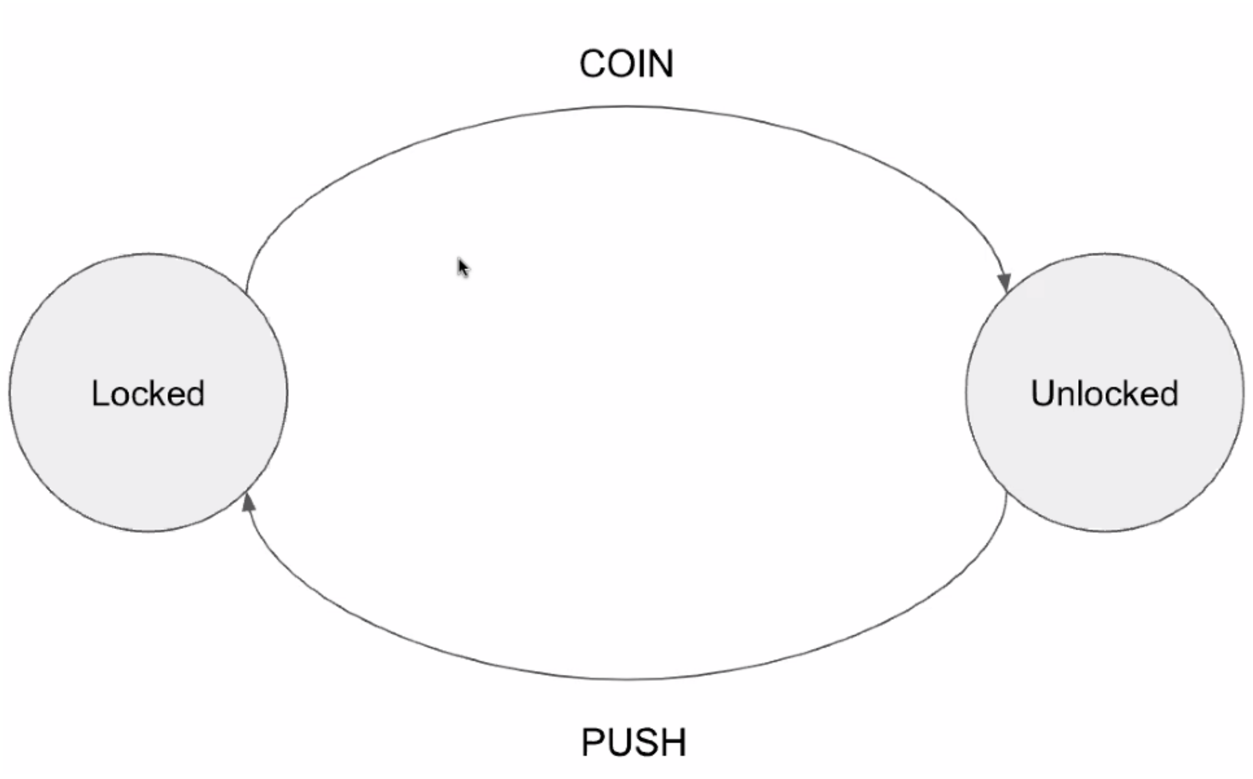
30p
Please note that barriers will not accept £1 – just 10p, 20p or 50p – we do not give change.
20p
10p
50p

Chris gave us 60 seconds to think of as many things that could happen as possible (can be weird too – he gave an example about aliens coming!). The point is that the issue with the world is that it is unpredictable and uncontrollable – there are an infinite number of possibilities!

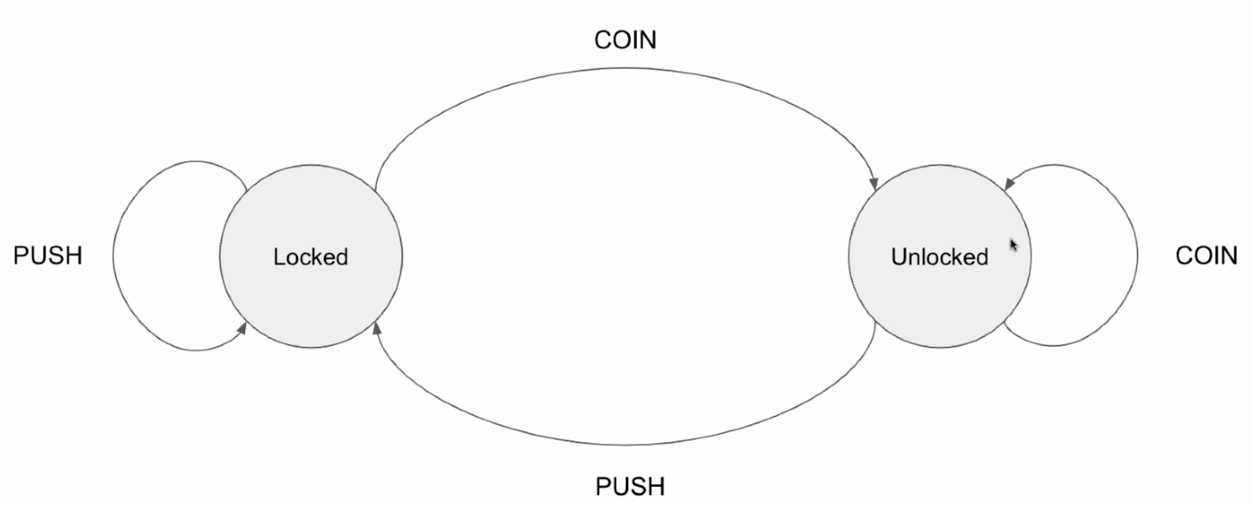
So how would we program this turnstile? As a programmer, we need to ignore ALL the possibilities and make it as simple as possible. But sometimes this will butt heads against reality. Computers are finite in terms of memory etc, but the universe is infinite.

In our turnstile example, we only need to concern ourselves with 2 things – whether it should be locked or unlocked. (This is an example of state).

How do we get from 1 state (locked) to the other (unlocked)? We only want 2 things to happen to allow this state to change – you pay (Coin) or push the turnstile.



Each action only changes the state once (e.g. paying doesn't change it back to lock). No matter how many times you push whilst it is locked, it won't change.



This is boiling the problem down to a finite number of states, with a finite number of actions to take to change the state. This is how we make things predictable!

Another example: Asking a user whether to go forwards or backwards. You wouldn't want to give them an input field to type forwards or backwards as they may spell it incorrectly, use wrong case, type in other things etc. So boil this down to 2 buttons – 1 called forwards and 1 called backwards.

So, we are trying to limit the possibilities that the user can take.

Finite state machines are all about reducing it down: State + action = new state
In above example state is locked and action is coin, so new state is unlocked. If state was unlocked and action is coin, new state is (still) unlocked.

We then applied this concept to making a To Do List app:

Chris's plan (starting simple – you can add in more functionality):

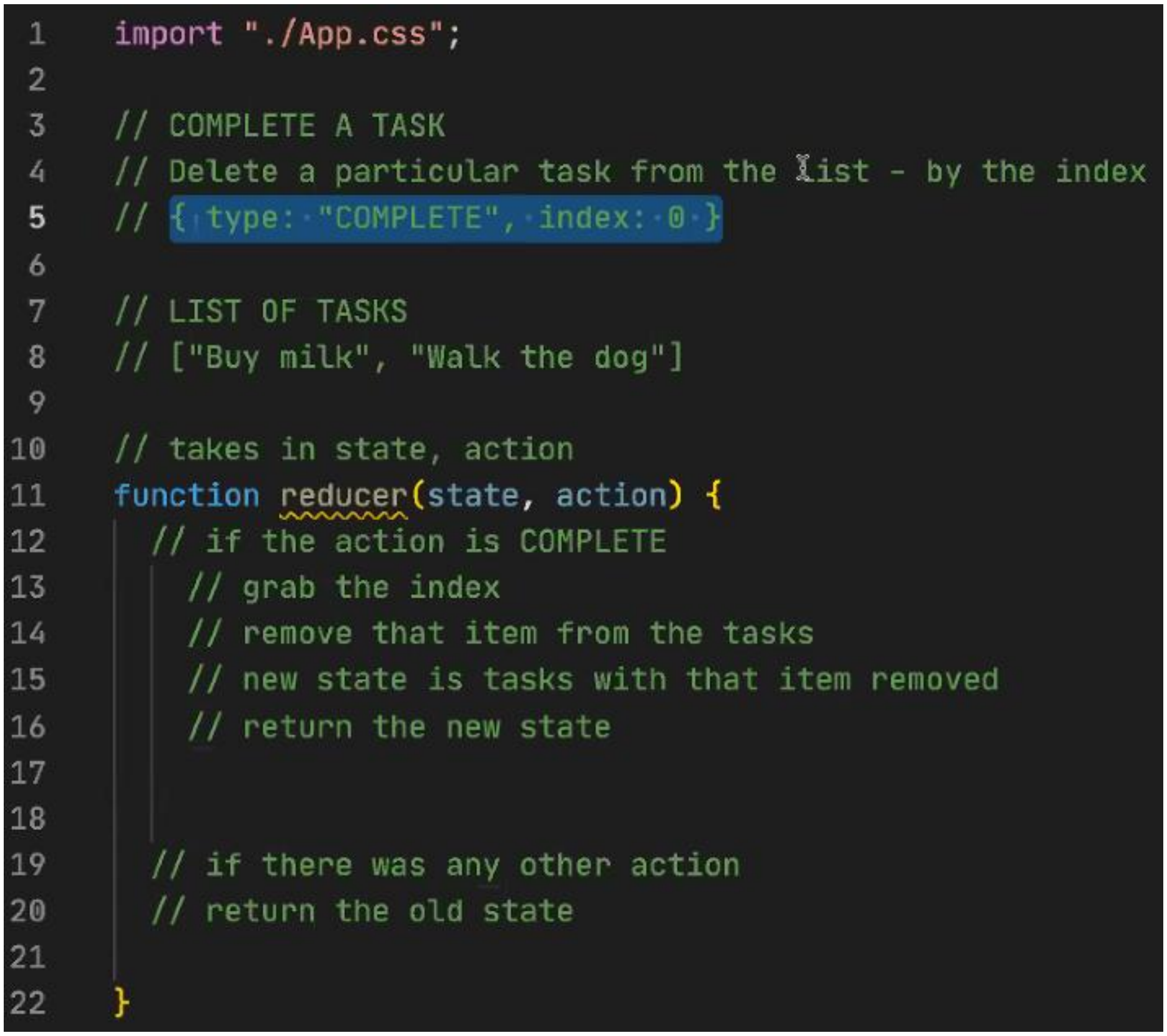
1st example of what we want to be able to do:

- Complete a task (this is an **action** - most important – whole point of a to-do list!)
- List of tasks (this is the **state** for the above action – probably going to look like an array of strings at its most basic e.g. ["Buy milk", "Walk the dog"])
- **New state** should be either ["Buy milk"] or ["Walk the dog"] (and if done again, then **new state** is empty array)

//Complete a task
Delete a particular task from the list – by the index (could use title but may give duplicates)
This will be done by adding a string of "COMPLETE"

//List of tasks
["Buy milk", "Walk the dog"]

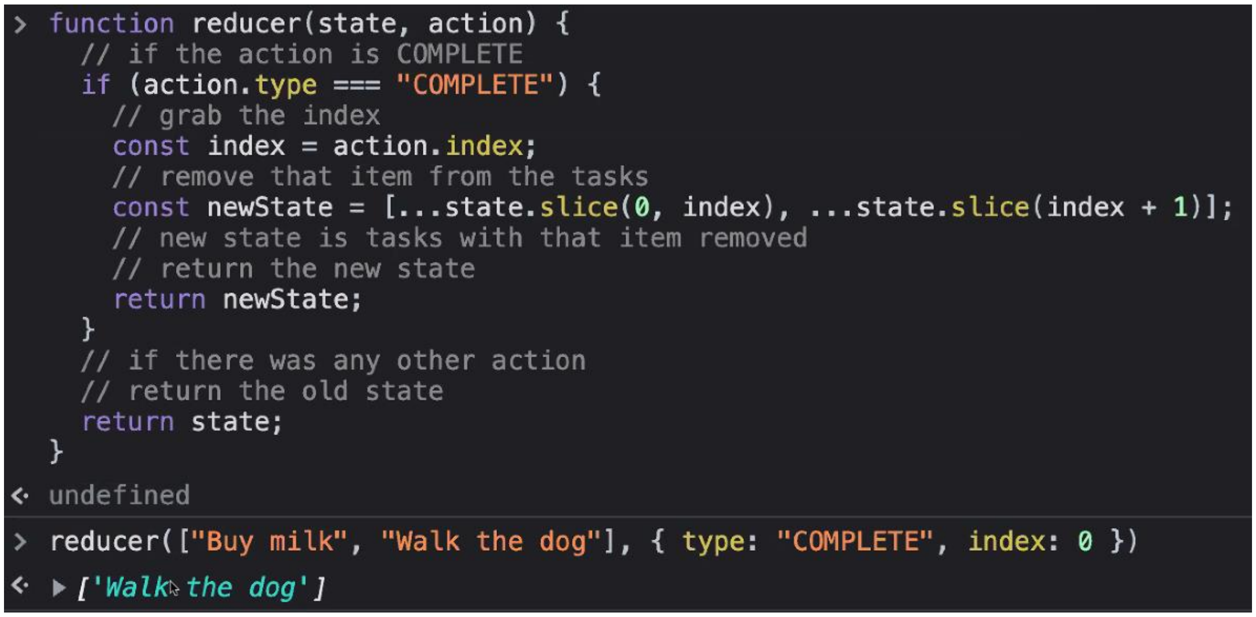
Then wrote a reducer function (doesn't have to be called reducer and params don't need to be called state and action) – that takes a state and action then returns a new state



```
 1  import "./App.css";
 2
 3  // COMPLETE A TASK
 4  // Delete a particular task from the list - by the index
 5  // { type: "COMPLETE", index: 0 }
 6
 7  // LIST OF TASKS
 8  // ["Buy milk", "Walk the dog"]
 9
10  // takes in state, action
11  function reducer(state, action) {
12    // if the action is COMPLETE
13      // grab the index
14      // remove that item from the tasks
15      // new state is tasks with that item removed
16      // return the new state
17
18
19    // if there was any other action
20    // return the old state
21
22  }
```

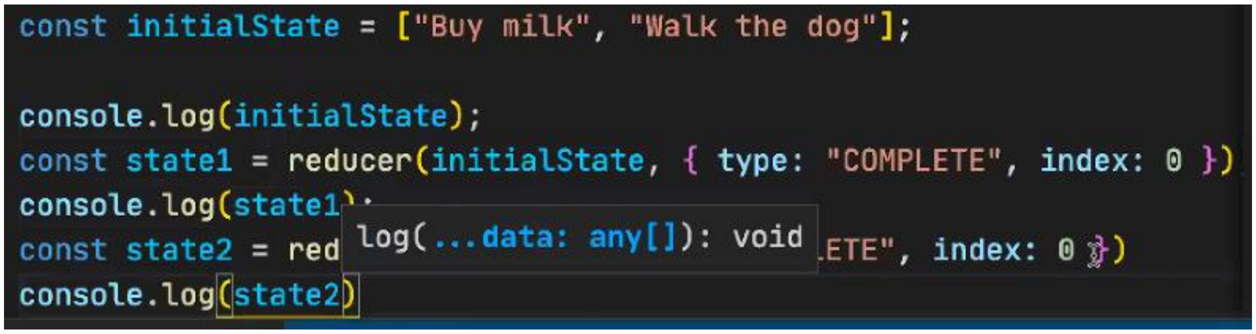
Line 5 is an example using object representation of what the action is, and gives us the index.

Wrote the function then used it in the console:



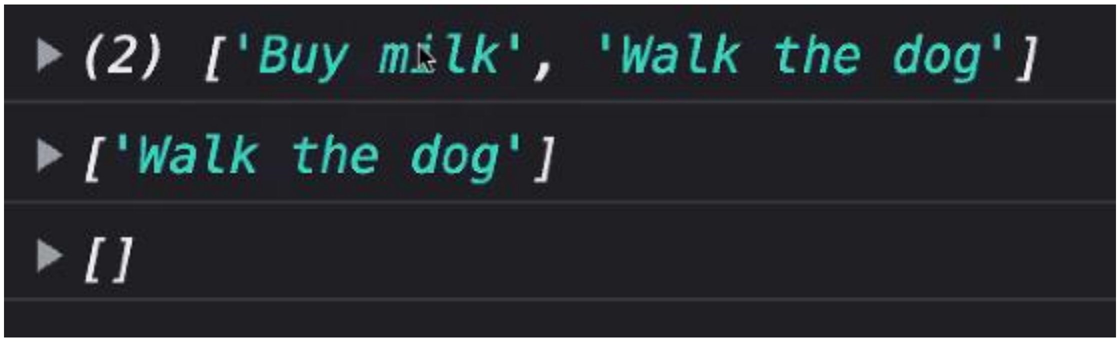
```
> function reducer(state, action) {
    // if the action is COMPLETE
    if (action.type === "COMPLETE") {
      // grab the index
      const index = action.index;
      // remove that item from the tasks
      const newState = [...state.slice(0, index), ...state.slice(index + 1)];
      // new state is tasks with that item removed
      // return the new state
      return newState;
    }
    // if there was any other action
    // return the old state
    return state;
  }
<· undefined
> reducer(["Buy milk", "Walk the dog"], { type: "COMPLETE", index: 0 })
<· ▶ ['Walk the dog']
```

Doing it in the code:



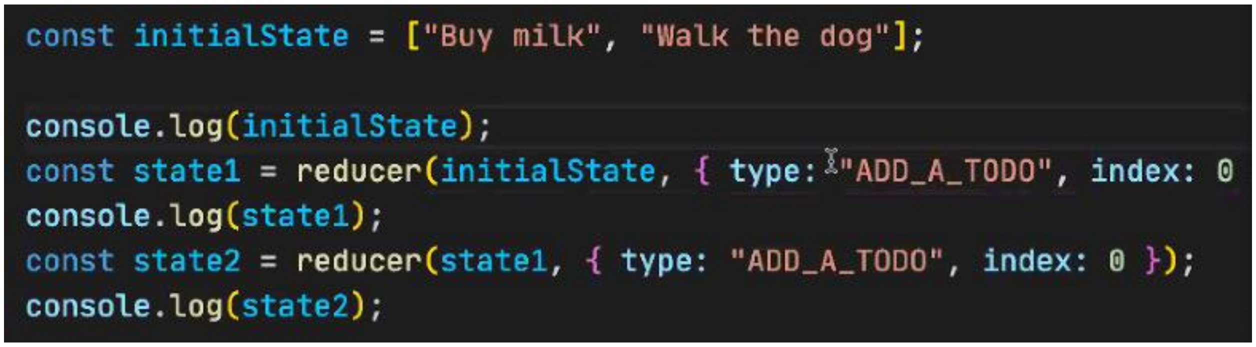
```
const initialState = ["Buy milk", "Walk the dog"];

console.log(initialState);
const state1 = reducer(initialState, { type: "COMPLETE", index: 0 })
console.log(state1);
const state2 = red log(...data: any[]): void ETE", index: 0 })
console.log(state2)
```

Result in the console:



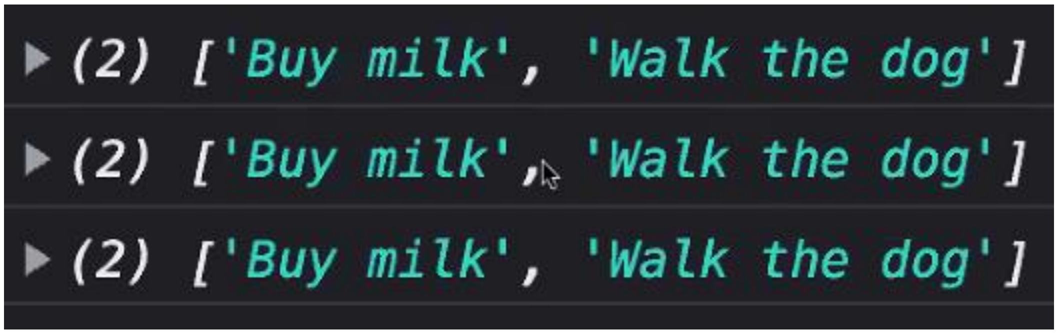
```
▶ (2) ['Buy milk', 'Walk the dog']
▶ ['Walk the dog']
▶ []
```

Give it a different action (such as Add a todo):



```
const initialState = ["Buy milk", "Walk the dog"];

console.log(initialState);
const state1 = reducer(initialState, { type: "ADD_A_TODO", index: 0
console.log(state1);
const state2 = reducer(state1, { type: "ADD_A_TODO", index: 0 });
console.log(state2);
```

Nothing happens to state in the console:



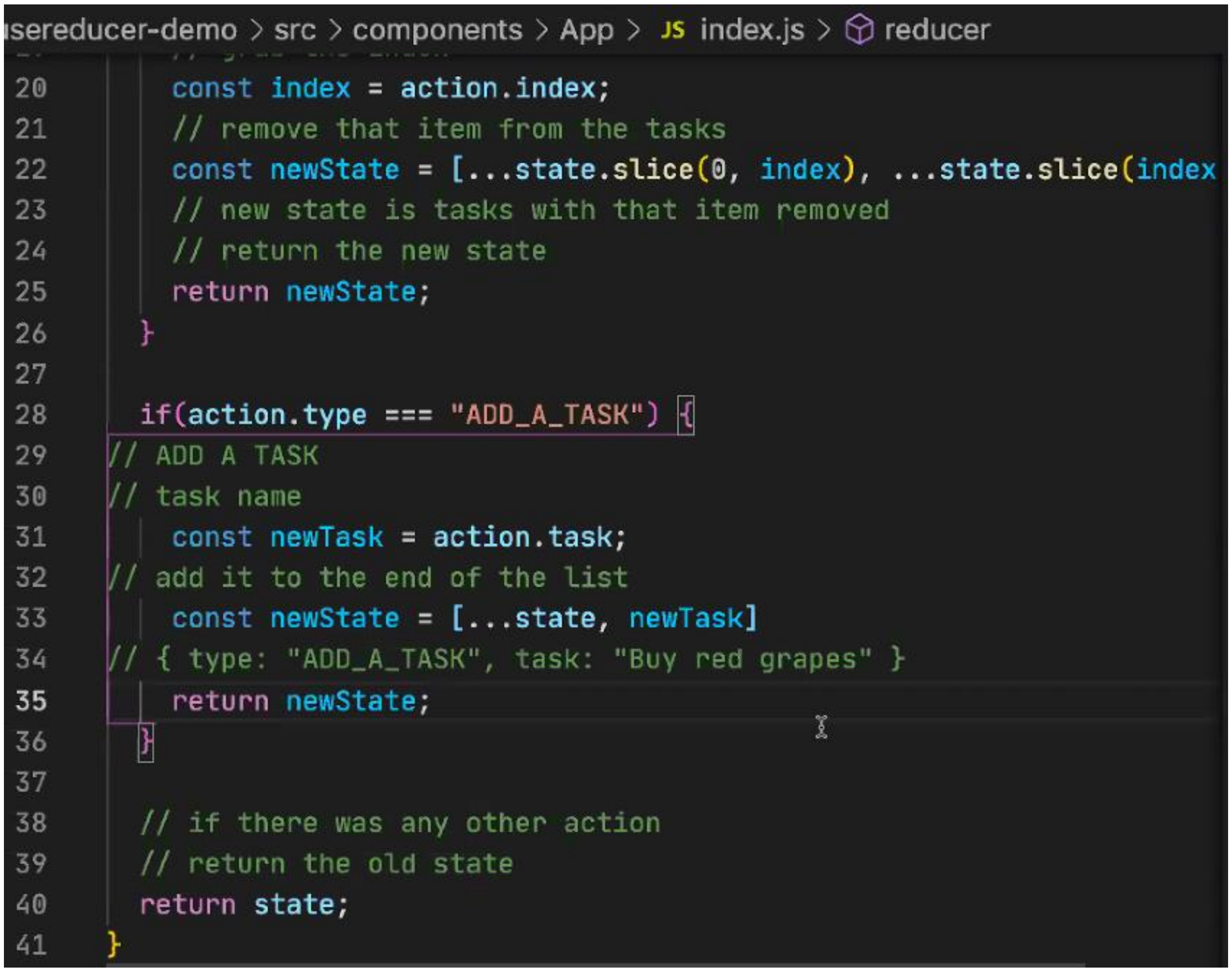
```
▶ (2) ['Buy milk', 'Walk the dog']
▶ (2) ['Buy milk', 'Walk the dog']
▶ (2) ['Buy milk', 'Walk the dog']
```

But this is really powerful as there are no bugs, it just didn’t do it!

So can then add another if check for action.type=”ADD_A_TODO”:

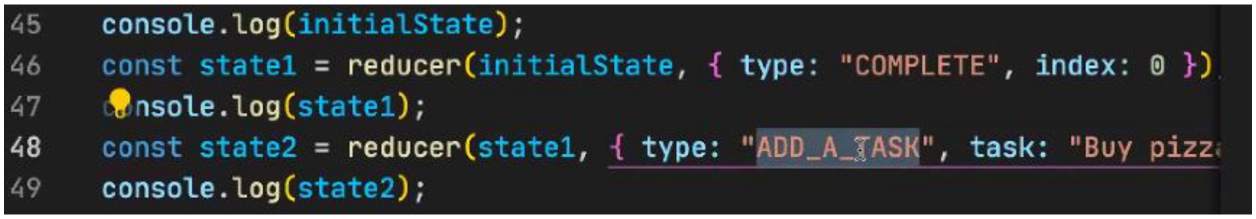
//Add a task Plan
Need task name
Add it to the end of the list
{type: “ADD_A_TASK”, task: “Buy red grapes”}

Writing the code:


sereducer-demo > src > components > App > JS index.js > reducer

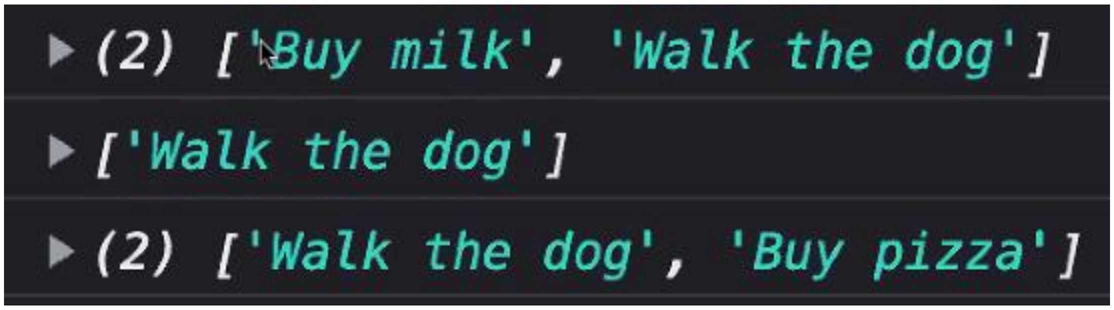
```
        const index = action.index;
        // remove that item from the tasks
        const newState = [...state.slice(0, index), ...state.slice(index
        // new state is tasks with that item removed
        // return the new state
        return newState;
      }

      if(action.type === "ADD_A_TASK") {
    // ADD A TASK
    // task name
        const newTask = action.task;
    // add it to the end of the list
        const newState = [...state, newTask]
    // { type: "ADD_A_TASK", task: "Buy red grapes" }
        return newState;
      }

      // if there was any other action
      // return the old state
      return state;
    }
```

Modified the code to display in the browser:



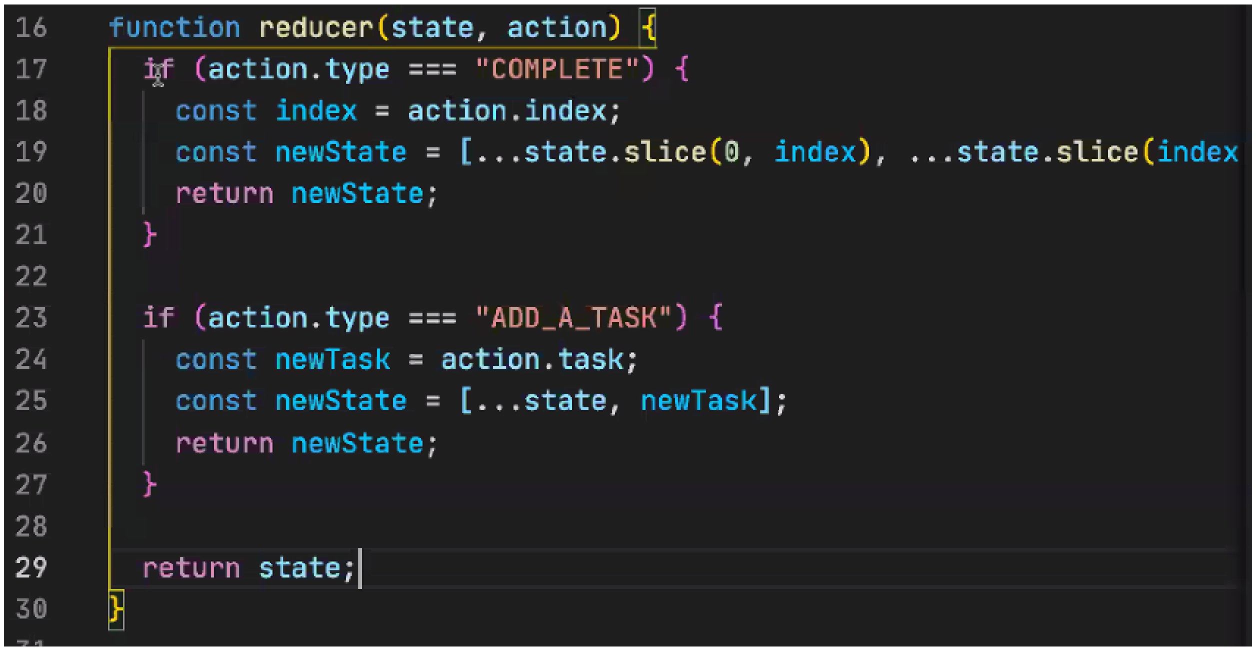
```
console.log(initialState);
const state1 = reducer(initialState, { type: "COMPLETE", index: 0 })
console.log(state1);
const state2 = reducer(state1, { type: "ADD_A_TASK", task: "Buy pizz
console.log(state2);
```

Console result:



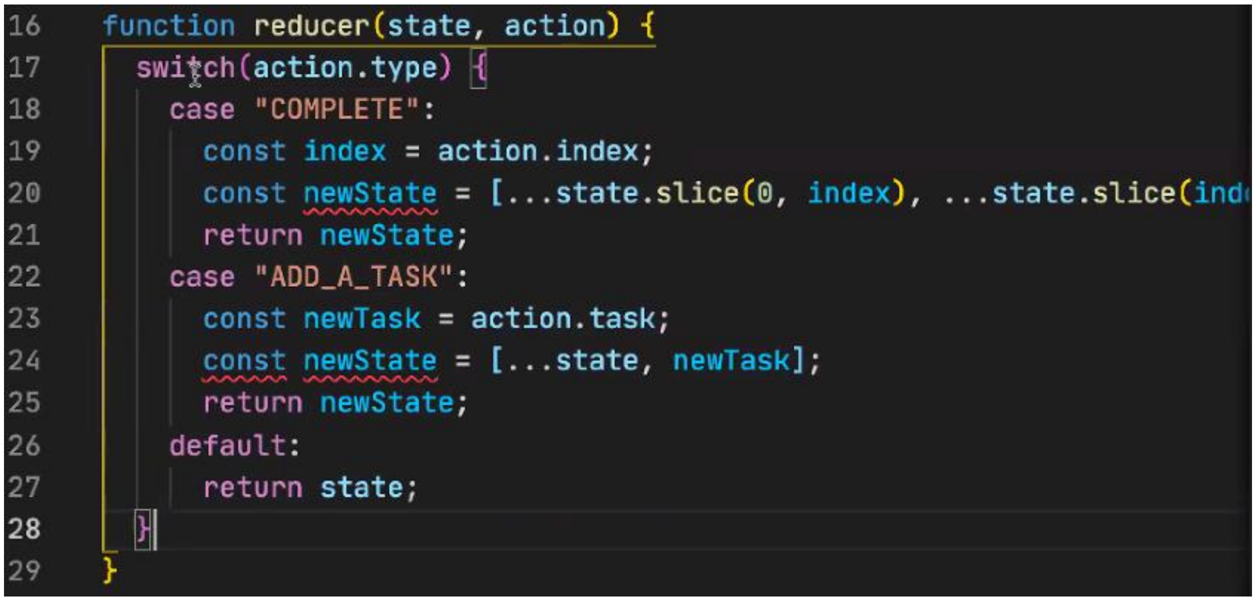
```
▶ (2) ['Buy milk', 'Walk the dog']
▶ ['Walk the dog']
▶ (2) ['Walk the dog', 'Buy pizza']
```

Whole code:



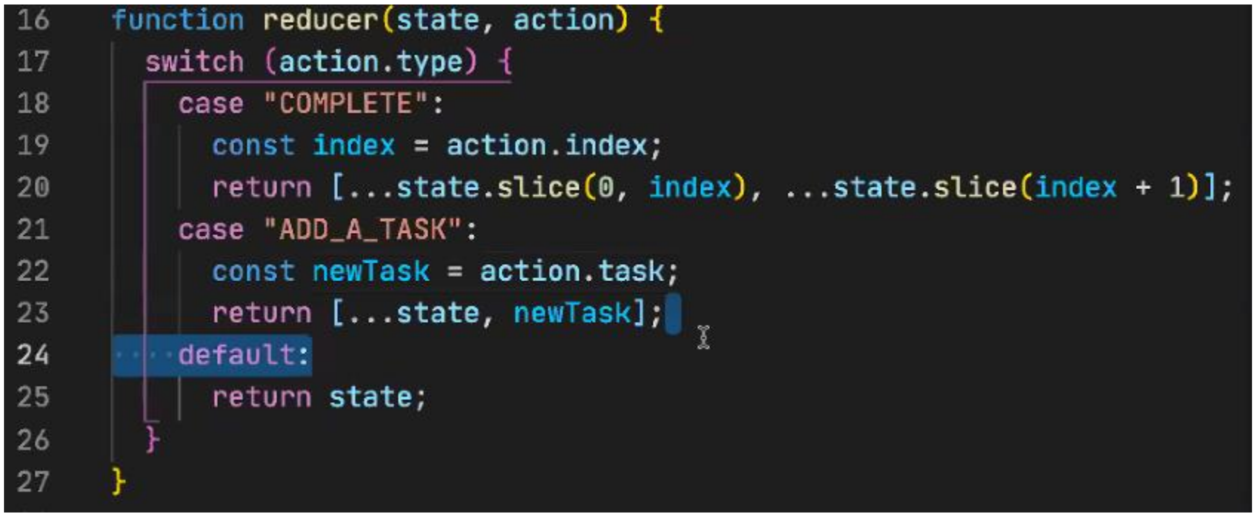
```
16  function reducer(state, action) {
17    if (action.type === "COMPLETE") {
18      const index = action.index;
19      const newState = [...state.slice(0, index), ...state.slice(index
20      return newState;
21    }
22
23    if (action.type === "ADD_A_TASK") {
24      const newTask = action.task;
25      const newState = [...state, newTask];
26      return newState;
27    }
28
29    return state;
30  }
```

Convention is to use SWITCH statement instead of if check, because you are always checking the same thing. For each case in the switch, you need to calculate and return some next state:



```
16  function reducer(state, action) {
17    switch(action.type) {
18      case "COMPLETE":
19        const index = action.index;
20        const newState = [...state.slice(0, index), ...state.slice(ind
21        return newState;
22      case "ADD_A_TASK":
23        const newTask = action.task;
24        const newState = [...state, newTask];
25        return newState;
26      default:
27        return state;
28    }
29  }
```

It doesn’t like that we have multiple things named the same thing in the same code block (line 20 and 23). Easiest way to get around this is to return the state directly, rather than naming it each time (line 20 and 23):



```javascript
function reducer(state, action) {
  switch (action.type) {
    case "COMPLETE":
      const index = action.index;
      return [...state.slice(0, index), ...state.slice(index + 1)];
    case "ADD_A_TASK":
      const newTask = action.task;
      return [...state, newTask];
    default:
      return state;
  }
}
```

Chris' little gems:

Having the diagram as outlined above allows you to break the task down between your team and plan the code better.

Reducers are really testable because you know what you are expecting to get back!

“You don’t know what you don’t know” – this is levels above what we need to know right now, but we are being shown this so we can at least remember the concept and google for it later!

<u>**Why are reducers functions useful?**</u>

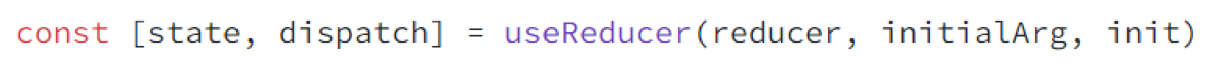
You can do exactly the same with state and useState. But reducers are useful when multiple pieces of state that interact with each other – because it helps simplify the problem.

How to get this in to React with useReducer:

React has useReducer hook. So we will use their example and embed it in ours:

https://beta.reactjs.org/apis/usereducer#adding-a-reducer-to-a-componet

Syntax is:



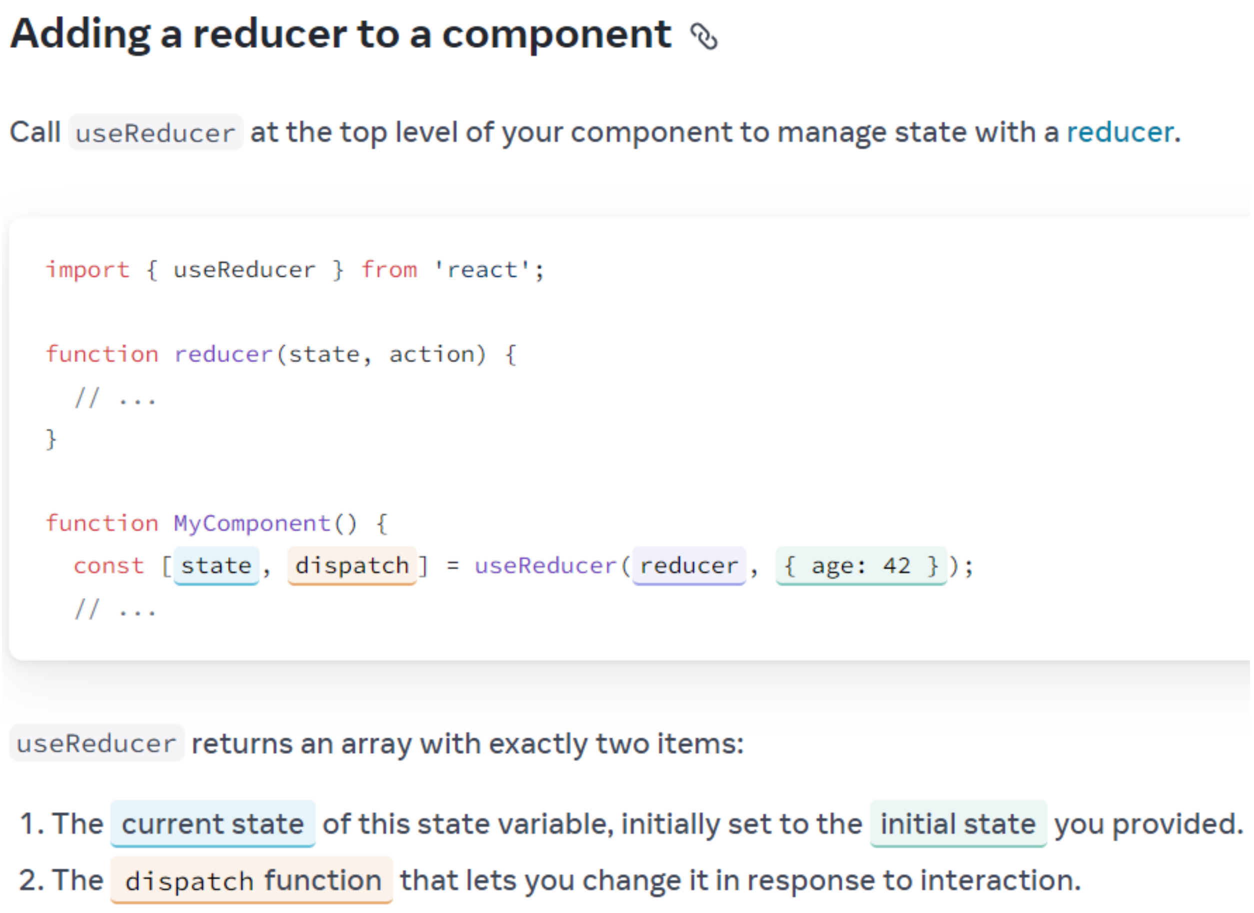
```
const [state, dispatch] = useReducer(reducer, initialArg, init)
```

Needs to be imported at top of file like useState and useEffect.
Works in a similar way to useState – state holds current state (that's passed in as "initialArg"). You then use the dispatch function to update the state – just tell it what you want to perform.

Parameters

- reducer: The reducer function that specifies how the state gets updated. It must be pure, should take the state and action as arguments, and should return the next state. State and action can be of any types.
- initialArg: The value from which the initial state is calculated. It can be a value of any type. How the initial state is calculated from it depends on the next init argument.
- **optional** init: The initializer function that specifies how the initial state is calculated. If it's not specified, the initial state is set to initialArg. Otherwise, the initial state is set to the result of calling init(initialArg).

Dispatch calls the reducer function, passing in the state and the action. Is written at top of your component function:



Adding a reducer to a component

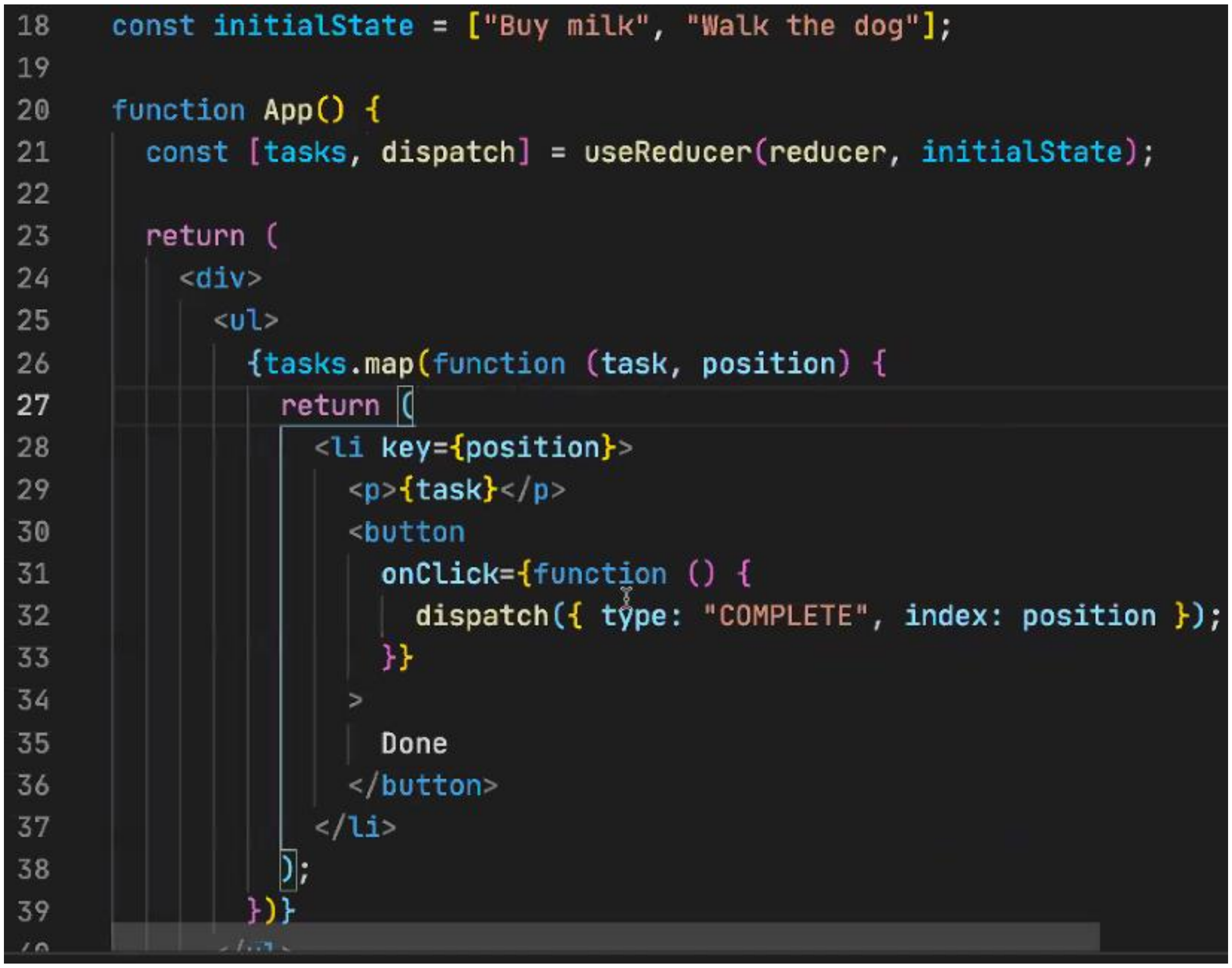
Call `useReducer` at the top level of your component to manage state with a reducer.

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

function reducer(state, action) {
  // ...
}

function MyComponent() {
  const [state, dispatch] = useReducer(reducer, { age: 42 });
  // ...
```

`useReducer` returns an array with exactly two items:

1. The current state of this state variable, initially set to the initial state you provided.
2. The `dispatch` function that lets you change it in response to interaction.

With our example:



```
18  const initialState = ["Buy milk", "Walk the dog"];
19
20  function App() {
21    const [tasks, dispatch] = useReducer(reducer, initialState);
22
23    return (
24      <div>
25        <ul>
26          {tasks.map(function (task, position) {
27            return (
28              <li key={position}>
29                <p>{task}</p>
30                <button
31                  onClick={function () {
32                    dispatch({ type: "COMPLETE", index: position });
33                  }}
34                >
35                  Done
36                </button>
37              </li>
38            );
39          })}
```

Line 21 sets up the Reducer, passing in the tasks array, and the reducer function that Chris wrote earlier. This reducer function is called via line 32 using dispatch – we only need to pass in the action (using the object we used earlier), React will pass in the current state automatically.

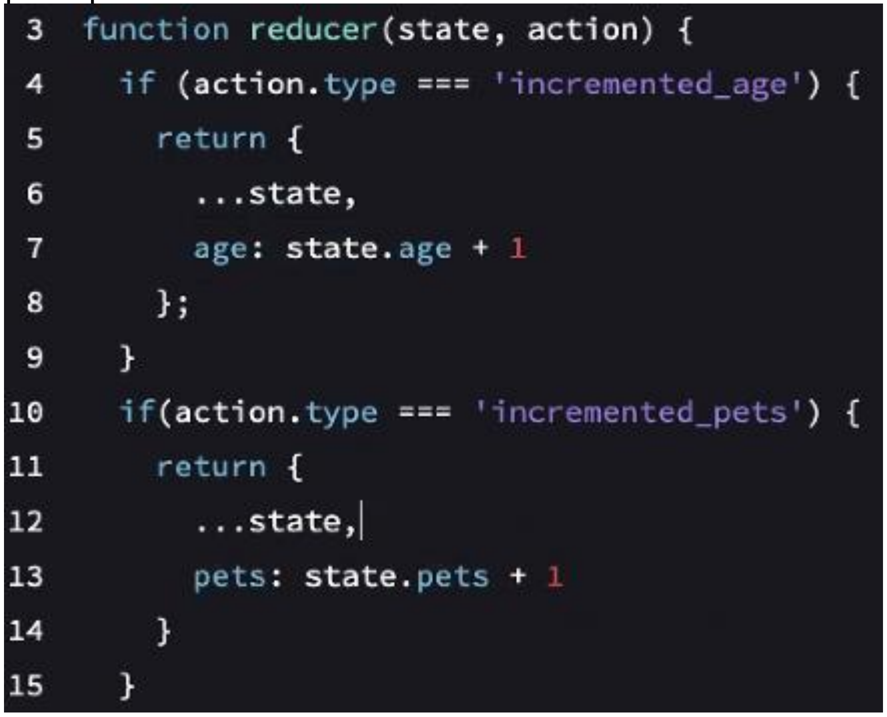
N.B. This code is using the index of the array item as the key. **Don't do this!** (see React Resources for link to article on why not).

Chris' little gems:

- You can have multiple reducers, but if you get to that stage, you may have put too much complexity in one component!!!
- Likewise with state – if you are using 10 states, then you probably should be using a reducer instead of useState!
- Reducers give us complex state management and fewer bugs
- **Dispatch is asynchronous!**

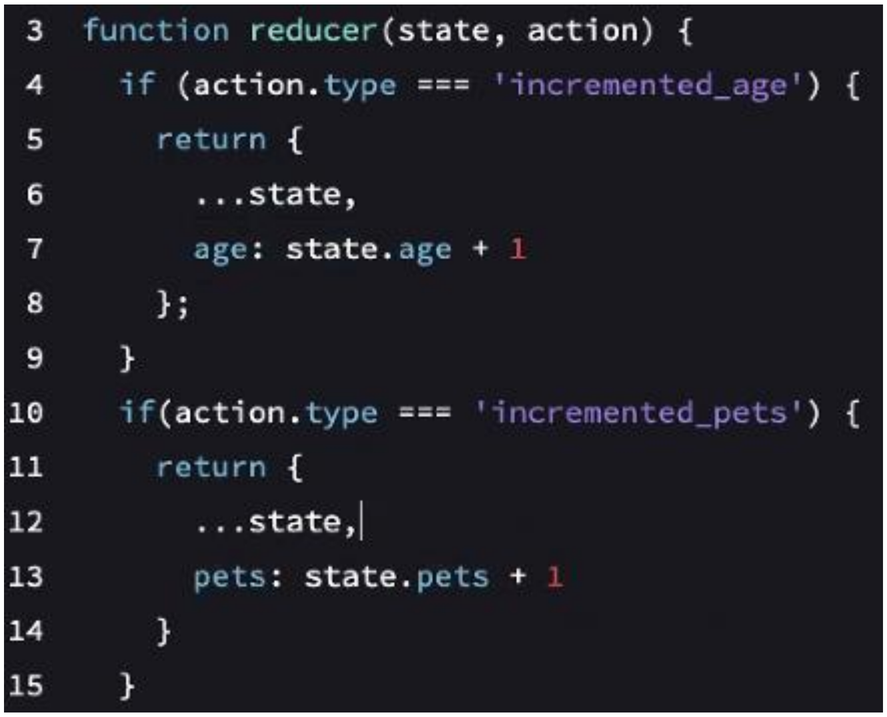
Points from recapping the docs together

- All hooks work in a similar way to useState – if you can use that, then you can pick up the others.



```
3  function reducer(state, action) {
4    if (action.type === 'incremented_age') {
5      return {
6        ...state,
7        age: state.age + 1
8      };
9    }
10   if(action.type === 'incremented_pets') {
11     return {
12       ...state,
13       pets: state.pets + 1
14     }
15   }
```

- (Especially when you have a lot of state) Use the … spread operator first to pass in the old state, then only pass in the values to be updated. This updates it because you are overwriting the key (age on line 7, pets on line 13, so make sure you spread the state first!): Make sure you include whatever extra info the reducer needs to take the action when calling dispatch (such as the new name to update):

- 

```
function reducer(state, action) {
  if (action.type === 'incremented_age') {
    return {
      ...state,
      age: state.age + 1
    };
  }
  if(action.type === 'incremented_pets') {
    return {
      ...state,
      pets: state.pets + 1
    }
  }
```

- Immer is a library you can use to type mutable code and it will do it immutably – doesn’t do anything you can’t do yourself and you should get in to the habit of writing “correct” code anyway!
- You can pass in a third argument with an initialiser function to the useReducer hook, if the initial state is being created by a function call (the 2nd argument is passed in to the function, so make it null if initialiser function takes no arguments)